Extensible Type Specifications for RDF and XML Schemas

Frank Olken
Lawrence Berkeley National Laboratory
September 11, 1998

DRAFT 1.0

Introduction

This document addresses issues of extensible type specifications for use in XML and RDF schemas, i.e., schemas which describe information encoded as either XML or RDF documents. XML documents are described by XML schemas such as XML-Data and DCD (Document Content Definitions). RDF documents are described by W3C RDF schemas.

We are particulary concerned with two issues:

- decomposing the descriptation of basic types
- extensibility of type specifications

A Common Type System for XML and RDF?

The immediate origin of this work was a perceived need by the RDF Schema Working Group for an explicit type system for use By RDF Schemas of RDF documents. Meanwhile Microsoft, et al. had proposed XML-Data as an attempt to recast DTDs into XML and to provide a type system for XML documents. Subsequently, Microsoft, et al. have proposed DCD (Data Content Definition) which is similar in intent to XML-Data. However, DCD encodes the schema definition as an RDF document.

In various discussions, it has become clear that many W3C participants believe that both XML and RDF need extensible type systems, and that it would be preferable if these two type systems were the same, or at least had the same basic types. We are proceeding under this assumption and are therefore we are attempting both sets of requirements.

In particular, we view RDF as the basis for our proposed type specifications, e.g., as does DCD.

Basic Type Specifications

A basic type specification is a tuple of type attributes, which specify the characteristics of basic data types:

- Semantic Class: integer, real number representation, string, date, date-time (?)
- Storage length: "n" octets
- Transfer representation: e.g., floating point formats, character set encodings ....
- Order relations: partial, total, none ....

A Taxonomy of Type Extensibility

- **Subtyping** - e.g., more restrictive types, inheritance

  Subtyping constructs a new type which is a further restriction of a data type, e.g., positive integers are a subtype of integers. Subtyping is one of the most common type extensions. It is supported in most programming

languages and object oriented modeling methodologies, usually via "isa" relationships, and some inheritance mechanism. Note that range constraints (esp. of basic numeric types) are very common.

Inheritance of type specifications (and methods) is a major area of dispute, especially as regards single vs. multiple inheritance. We believe that multiple inheritance is quite important for concise specification of data type constraints.

- **Composite Types** - a.k.a. collection types

  Composite types are constructed via some sort of aggregation of more elementary types. Relational databases have sets or bags of tuples (aggregation) of basic types. Object oriented programming languages and DBMSs have additional composite type constructors:
  - aggregation (a.k.a. tuple, struct or record)
  - set of (no duplicates)
  - bag of (duplicates allowed)
  - sequence of (a.k.a. list)
  - vector of (single dimension array)
  - array of (possibly multidimensional)

  There is also the question of whether such composite types can be constructed recursively.

  Note that XML does not directly specify the type of composite structure. RDF currently supports some (bag, ...) of the type constructors, but not others.

- **Extensible type specifications**

  Two types of extensible type specifications are of interest:
  - additional attribute values, e.g., new character set encodings
  - additional type attributes, e.g., measurement units or dimensionality, or coordinate system

  Note that the second sort of type specification extensibility is typically not provided by most type systems. These are (effectively) new type attributes which are orthogonal to existing type attributes.

  There are also funny sorts of composite types (aggregates) used to denote various sorts of measurements such as length (feet and inches), weight (pounds and ounces), or latitude (degreees, minutes, seconds, north/south). Note that these composite constructions are equivalent to simple scalars (plus units). Arguably, they should be treated as alternative representations of the scalar forms. In any case, they are ubiquitous in electronic commerce, GIS, etc. and need to be addressed at some point.

Types as Objects

In most languages types are not full rank objects, and no computations or queries on them are possible. We believe that treating types as full rank objects will enhance extensibility of the type system and facilitate implementation of extensible type systems. Treating types as objects (resources in RDF jargon) should fit nicely into RDF. Potential ramifications of such a decision should be carefully discussed, especially with members of the programming language communities, e.g,. ML, which have experience/knowledge of such practices.

Type Naming

Do we allow unnamed types: e.g., composite or subtypes? Or do we require all types to be named explicity? This has implications for specification of recursively constructed composite types.

Scope of Type Names

What is the scope of named types? global, XML namespace, or within enclosing type declarations?

Syntax issues

- **Use of XML arguments vs. XML elements**
- **Use of flat DTD-like grammar specifications vs. nested hierarchical type structure specifications.**
- **Relation to RDF**

Exclusive use of XML elements seems to be a cleaner, more systematic syntax and more readily extensible than the use of XML attributes. However, it is typically more verbose. Note that XMI (the OMG XML metadata interchange format) uses data elements solely (?). However, DCD (Data Content Definition) allows either encoding.

Nested specification of types follows conventions widely employed in programming languages, often these nested types are not named (but they could be). The flat grammatical approach seems less natural to programmers accustomed to nested hierarchical specifications of data types.

We believe that the factorization of basic type specifications into multiple attributes can be easily encoded in RDF, by treating a "type" as a resource, described by multiple "properties". As noted above, we view RDF (rather than raw XML) as the basis for our proposed syntax. This is the position of the DCD Note.

ISO 11404 - Language Independent Data Types

ISO Standard 11404 on Language Independent Data Types addresses many of the issues raised in our paper. We believe that it should be examined as a possible starting point for development of a type system for XML/RDF. In particular, ISO 11404 makes some attempt to decompose the specification of basic types into a set of attributes (semantic type, range constraints, storage representation, etc.)

ISO 11404 was originally developed for use by various programming language, database, and data exchange standards efforts in an attempt to bring about some convergence on data type standardization. It includes a large number of base types, and a number of type constructors.

The ISO 11404 specificaton can be found at ..... at least for W3C members (and other standards developers). Because the material is copyrighted, we are as yet unable to provide legal public access to the document online. Hardcopies are available from ANSI in the United States. We have begun discussions with ISO concerning public (web) release of this standard.

External Type System Compatibility

We believe it important to be (largely) interoperable with the most popular existing type systems used to encode data. Examples include SQL datatypes, ASN.1 datatypes, C, C++, Java, ODMG data types, ... We are primarily concerned (at present) with congruence of basic types: integer, string, etc.

Below we give brief descriptions of each type system. /

- ISO11404
- ASN.1 - an ISO data interchange standard
- SQL 2
- SQL 3?
- Java

Bibliographic Resources on Type Systems

- OO Type Theory by Laurent Dami
- Work related to "A Theory of Objects" by Cardelli, et al.
- A Theory of Objects by Luca Cardelli, et al.
- The Type Forum a mailing list on type theory

Maintained by Frank Olken at Lawrence Berkeley National Laboratory. olken@lbl.gov Last updated: September 11, 1998

W3C

NOTE-MCF-XML

Meta Content Framework Using XML

Submitted to W3C 6 June 97

This document is available at:
http://www.w3.org/TR/NOTE-MCF-XML-970624
Previous Version:
http://www.w3.org/TR/NOTE-MCF-XML-970606
Editors:
R.V. Guha (Netscape Communications) <guha@netscape.com>
Tim Bray (Textuality) <tbray@textuality.com>

Status of this Document

This document is a NOTE made available by the W3 Consortium for discussion only. This indicates no endorsement of its content, nor that the Consortium has, is, or will be allocating any resources to the issues addressed by the NOTE.

This document is a submission to W3C from Netscape Communications. Please see Acknowledged Submissions to W3C regarding its disposition.

Abstract

This document provides the specification for a data model for describing information organization structures (metadata) for collections of networked information. It also provides a syntax for the representation of instances of this data model using XML, the Extensible Markup Language.

Table of Contents

Appendices

---

1. Introduction

This document gives a complete description of the MCF data model and its syntactic expression. It is not tutorial in nature; a companion document, An MCF Tutorial, exists to serve that purpose.

1.1 History and Motivation

The need for machine-usable descriptions of collections of distributed information is increasing rapidly. There have been a number of proposals in the recent past that have made significant steps toward this goal, including HotSauce MCF, CDF, PICS, and WebCollections.

The existence of multiple proposals reflects the fact that this type of information is needed for multiple purposes, and that there are many groups interested in its availability and use. This diversity of effort is reflected in a diversity of terminology; discussions have been couched in terms of "metadata", "typing", "schemata", "labels", and "collections," while all in fact dealing with the same underlying constructs and problems.

We believe the following principles to be central to making progress in this area:

1. There is no useful distinction between the representational needs of data and metadata. The *kinds of information* that need to be represented in metadata and data are very similar. Furthermore, every item of information, without exception, is likely to be regarded by some applications as ancillary and never to be displayed, and by others as core content that needs to be formatted, printed, or searched.
2. For interoperability and efficiency, schemata designed to serve different applications should share as much as possible in the way of data structures, syntax, and vocabulary.

The consequence of the first principle is that it is simply incorrect to reserve any special representation for use just in "metadata".

The second principle is what really drives this proposal. It is inevitable that there will be a plethora of classes of information about information; note some of the examples listed above. If they share a common syntax, this is good, but it is not enough. For example, suppose a mature commercial word processor package were to offer a "save as XML" format, which exported an XML representation of its internal document data structures and attributes. While marginally more open than the processor's native format, this would not be of any substantial use, because to operate on this file would de facto require the use of the program which generated it.

To a certain extent this is inevitable - in many cases, data created for the purposes of a particular application will contain items that are only meaningful to that application. But the situation can be greatly improved. If information about information can share a common data model and vocabulary, it will be possible to build software to query and manage it without knowing all the schemata in advance.

1.2 The Basis of Meta Content Framework

In this document, we draw upon the features provided in the other proposals mentioned above, and on other work in this area, to develop a single data model and corresponding interchange format which can be used for many purposes, including for example

- describing the structure of web sites or a set of channels
- threading email
- PIM functions
- distributed annotation and authoring
- exchanging commerce-related information such as prices, inventories, and delivery dates

Meta Content Framework (henceforth referred to as MCF) is a structure description language. The field of structure description languages is well understood and it is not our desire to reinvent any of it. Our goal is to select the portions of it that are required for our task. One benefit of this approach is the ready availability of tools and algorithms for manipulating MCF.

We abstract an information organization structure as a Directed Labelled Graph (DLG). DLGs are well understood and as far as possible, we will use the terminology that is standard to the treatment of DLGs. In MCF, relationships between objects are represented in an unsurprising way by DLG arcs. DLG arc labels are themselves objects which participate in relationships.

New kinds of data appear on the web routinely. It should be possible to extend MCF dynamically to accommodate them. Furthermore, the list of potential applications for MCF is open-ended and each application might wish to add and use its own kinds of metadata. Though an application might associate arbitrary semantics with the new labels, it would be highly desirable if some significant portion of these semantics could itself be expressed with MCF. In light of these requirements, using DLGs, we include a simple, extensible type system as part of MCF.

2. The MCF Data Model

2.1 Labels, Nodes, and Arcs (or PropertyTypes, Nodes and Properties)

An MCF database is a set of Directed Labelled Graphs, comprising:

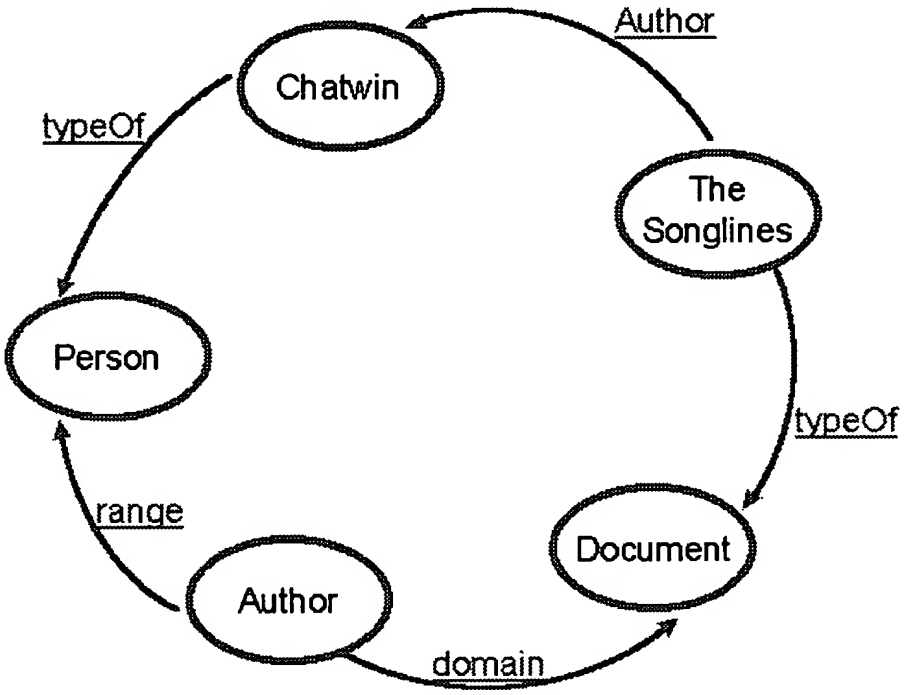
1. a set of labels, also referred to as property types
2. a set of nodes
3. a set of arcs where each arc is a triple consisting of two nodes (the source and target) and a label. Arcs are also referred to as properties. Often, we will refer to an arc with a certain source as a *property of that source*. Similarly we will refer to the target of the arc as the *value of the property*.

In MCF, nodes can represent things like web pages, images, subject categories, channels, and sites. They can also represent "real-world" objects such as people, places, and events.

The arcs (properties) can represent characteristics such as *size* or *lastRevisionDate* of web pages, subject categories, etc., and also their relationships (such as hyperlinks, authorship or parenthood) to other objects.

Each property type is a node (but not all nodes are property types). So, if we had a property type *pageSize* that is used to specify the basic size of documents, we would also have a *pageSize* node. This node could itself participate in properties that help constrain and therefore specify the semantics of *pageSize*. We would for example specify that the *domain* of *pageSize* is *Document* and its *range* is *SizeInBytes* and that a document has exactly one *pageSize*. It could also have a property to provide human readable documentation of the intended semantics of *pageSize*.

The figure below illustrates some simple nodes (including some property types) and properties, illustrating that how properties can be attached to property types.



This self-description allows MCF to be its own schema definition language. This in turn allows MCF to be dynamically extended by an author or application.

2.2 Units and Primitive Data Types

A node can either be a primitive data type or a "Unit". The primitive data types are the same as the Java primitive data types. In addition, a DATE type should be supported by the low-level MCF machinery, because it is tricky to implement (beyond the reach of regexps, for example) and yet commonly available in operating system and compiler libraries, e.g. *java.util*.

The concept of "Unit" corresponds loosely to the Java concept of "Object".

2.3 The Set of Bootstrap Nodes

A small set of units with predefined semantics are assumed to exist in order to bootstrap the type system. These names are reserved in MCF and may not be used for any purpose other than that given here. Specifically, these are,

`typeOf`
: this is the PropertyType used to specify that the given object is of a certain type. A node can be the origin of multiple *typeOf* arcs; for example, the node for a person can simultaneously be *typeOf* Person, *typeOf* Golfer, and *typeOf* Doctor. Every unit has (at least implicitly) a *typeOf* property, since Unit is a type.

`Category`
: This corresponds to the concept of Class. The destination of *typeOf* arc has a *typeOf* arc which ends at Category (with the single exception of the node for "Category" itself).

`Unit`
: This is the most general Category. It is implicitly or explicitly the super class of all Categories (with the single exception of the node for "Unit" itself).

`domain`
: this PropertyType is used to specify the type constraints on a property, in particular of its origin node; the range of *domain* is Category.

`range`
: this PropertyType is used to specify the type constraints on a property, in particular of its destination node. The range of *range* is Category.

`superType`
: this PropertyType is used to indicate the superset relation between Categories. If A is the superType of B and X is typeOf B, then X is also implicitly typeOf A.

`PropertyType`
: this is the *typeOf* all property types/labels.

`FunctionalPropertyType`
: Certain property types behave like functions, i.e., there can be at most one arc of that type originating from a given node. e.g., *lastRevisionDate*. Such properties are *typeOf FunctionalPropertyType*.

`superPropertyType`
: a relation between two property types. If s1 is a *superPropertyType* of s2, then the existence of an s2 arc between nodes A and B implies that there is also an s1 arc between A and B. E.g., *biologicalParent* is a *superPropertyType* of *biologicalFather*.

`mutuallyDisjoint`
: a reflexive relation between two categories which implies that nothing can be an element of both these categories simultaneously. For example, the categories for the built-in types (int, float, etc) are all *mutuallyDisjoint*.

`name`
: this can be used to provide a string which names the object. An object may or may not have a name, but it is neccessary for property types and categories to have names. Furthermore, the names of Category and PropertyType units are constrained to be valid XML Names. The tokens "typeOf", "PropertyType", etc. are the names of the corresponding units listed here.

`description`
: a descriptive string used for human consumption.

`parent`
: is is the most generic relation. The *domain* and *range* are Units.

`Sequence`
: This category is a special convenience used to express sequences. It is normally expected to source a number of arcs whose labels are natural numbers sequentially increasing from 1; the targets of these arcs are the nodes which are to be considered sequenced.

`ord`
: (short for ordinal) is a property type not actually used in MCF, but which is reserved because the label is needed for the syntactic expression of MCF in XML.

`Property`
: is a reserved term to be used in future versions of this document as a Category that will enable us to optionally treat certain properties as first class units. This will allow us to represent meta-meta content such as the volatality of a certain piece of meta content. The terms *source* and *target* are also reserved for property types that will apply to *Property* to specify the source and target of the arc/property.

As a convention, property types are named beginning with a lower-case letter and other units with an upper-case letter.

Though it is possible for a source of MCF to only assume the basic bootstrapping vocabulary and define everything else it needs dynamically, for purposes of interoperability, it would be good to standardize the vocabulary for commonly used terms. This will also reduce the amount of information that needs to be transmitted. An appendix to this document proposes some items for this vocabulary (largely derived from existing standards such as the Dublin Core) for describing web content.

3. Representation of MCF

3.1 Syntax

Our goal is to provide an XML based syntax for representing MCF. XML aims to serve as a general purpose data representation language. One of the components of any adequate data representation language is a type system; MCF attempts to provide such a type system for XML.

MCF is expressed using XML syntax with a few conventions provided by this specification. The XML text describing the MCF (which may occur as a separate file or be embedded within HTML) is wrapped inside a block tagged <XML-MCF> and </XML-MCF>. All MCF blocks are well-formed XML.

Given XML's flexibility, a number of strategies could serve for expressing MCF structures in terms of elements and attributes; all would be essentially isomorphic. However, it seems likely that it will be common practice to use MCF to express a series of facts about some object, framed as arcs with that object as the source.

Thus, the source is expressed as a container element, with a series of child elements each representing an arc with that source (i.e., property of that source.) The element type of the source element is the name of unit's *category*. If the unit is an element of more than one category, the additional categories can be specified using *typeOf* property elements. The container element may be given a unique identifier, which is a string provided in the *ID* attribute of this container element.

The element type of each of the child elements is the PropertyType associated with that arc. If the destination of the arc is a primitive type, it is represented as the content of the element. If the destination is a unit, it is represented by using the attribute *UNIT* attached to the element. The value of the *UNIT* attribute (i.e., the reference to the unit) must match the unique identifier of a container element representing that object (please see section 3.4 where this constraint is somewhat loosened.)

If the direction of the arc needs to be reversed, i.e., the container element is the target of the arc (value of the property) and the unit refered in the *UNIT* attribute is the source of the arc, this can be done by using the attribute *inverse* with the value "true". The default value for this attribute is "false".

It is legal for a unit to not have any unique identifier. In this case, it is not possible for any element representing a property to reference it.

The unique identifier for a unit is just that and does not have any binding semantics about locations on the web. There may be many different locations associated with a unit. For example, a unit representing a web page could have different locations for its mcf block, the actual content, ratings, etc. A unit representing a person could have locations for her home page, email address, etc. These locations can be expressed by using the appropriate properties. However, we do allow for certain defaults (see section 3.4) that enable more compact representations.

Uses of unique unit identifiers (i.e., as the value of the *UNIT* attribute within property elements and as the value of the *ID* attribute within the container element) within an MCF block follow the rules of URLS and so they may be either absolute or relative to the baseURL of the MCF block within which they occur.

3.2 Linking to Schemata

The sharing and re-use of schemata is uncontroversially good. In order to avoid duplication, we propose use of the XML Hyperlink machinery to refer to externally-stored MCF blocks. While details of this syntax will have to wait for that specification to stabilize, the following examples contain references which should be at least suggestive.

Of course, when multiple schemata are in use, a namespace problem occurs. In the following examples, we use the syntax of the recent Layman/Bray proposal; but the namespace resolution mechanism is an orthogonal problem.

3.3 Processing MCF Blocks

If a program reading an MCF block encounters a semantic contradiction, the entire MCF block is to be considered as unreliable and information from it is not to be used. An example of such a contradiction would be two arcs originating from the same node, labelled with a PropertyType that has been declared a FunctionalPropertyType, or for example, assertions that some node is both *typeOf float* and *typeOf character*.

Note, however, that different MCF blocks, obtained from different sources, describing same object, may be inconsistent. The decision as to how this should be handled is highly application-dependent.

3.4 Special Idioms

Beyond the above, there are several special XML idioms available for convenience and compactness in representing certain properties.

3.4.1 Unit Identifiers

We mentioned earlier that the unique identifier for a unit is just that and does not have any binding semantics about locations on the web. Having said that, it would desirable to have a set of default rules that enable more compact representations. So, as a default, unless explicit values for the corresponding properties are provided, for objects addressable on the Web and which have a canonical URL, it is expected to be common practice to use the URL as the unique identifier.

One of the implications of this default is that not all the units referred to in an MCF block need to have unit descriptor containers in that block or even in blocks included in that block. For example, a web page might not have any explicit MCF unit container corresponding to it, and yet, by using the URL as a unique identifier, a table of contents could refer to the unit that *denotes* the page.

For implementation considerations, we impose the constraint that property types and categories should have explicit descriptors that occur in either the MCF block, or more typically, in an included block, before their first use.

3.4.2 parent

The *parent* property may be expressed by element inclusion. That is to say, a source container element may contain not only property elements but also other source container elements; the effect is exactly the same as as if the contained source container were standing alone and contained a *parent* property pointing at the containing element.

3.4.3 Sequence

A Sequence node may have Properties whose labels are just numbers, sequentially increasing from 1, whose range is the sequenced nodes. These are expressed in XML simply by replacing the numbers with the reserved property *ord*; the order in which these Property nodes appear in the XML entity corresponds to the numeric labels.

3.4.4 Namespace Prefixes

We would like the most common case to be very simple. In the most common case, there will be exactly one schema used and since there will not be any schema ambiguities, the author should not have to do any extra work related to

namespaces. Furthermore, even if additional schemata are introduced, if there is a primary schema, the additional work should only be proportional to the extent to which the additional schema is used. To enable this, we allow for the first of the imported schemata to not have any associated prefix. Top level unit description elements that do not explicitly use any namespace prefix are assumed to use this schema.

Concretely, names which are taken from the first schema referenced (via XML-link) in an MCF block do not require prefixes; names from all others do.

3.4.5 Structured Values

There are many cases where the value of a certain property that a node has (e.g., address) is the concatenation of the values of a set of other properties of that node (e.g., streetAddress, city, state, zip). It would be convenient to not have to repeat these values. To enable this, we allow such values to be nested, as illustrated in example 3.

3.4.6 Inheritance

One of the most common uses of MCF will involve a publishing agent describing the organizational structure and other metadata about its web site. Many of these pages will share a lot of common properties (such as their table of contents, authorship, copyright and legal notices, etc.) It would be highly desirable not to have to repeat these. To enable this, we tentatively provide a simple inheritance mechanism.

The inheritence is accomplished in the XML representation using the *inherits* element. This appears as a child of an element representing a Category; it has an attribute named *propertytype*, and a value, provided in the usual way either with a *unit* attribute or in the *propertytype* element content. The effect is that all nodes with *typeOf* the Category are considered to have a property whose source is that node, whose associated property type is the value of the *propertytype* attribute of the *inherits* element and whose value is the value of that *inherits* element. Please see example 3 for an illustration of the use of this feature.

There is no direct analogue in the DLG representation; the XML expression asserts the existence of a (potentially large) number of arcs in the DLG.

3.5 HTML and MCF

For HTML pages, presumably the HTML *LINK* element would be used to associate MCF blocks that provide metadata about that page.

4. Examples

4.1 The Acme Content Company Web Site

The following examples contains information about the Acme Content Company web site that can be used for diverse purposes. For example,

- a robot could use it to determine which portions of the site to index.
- a browser could use it to present a site map.
- a push client could use it to periodically download portions of the site.
- the rich information here could be used by a search engine to provide better search (filters, concept based searches, etc.)

Given below are a sequence of three examples, each building on the other.

1. The first example provides a very simple table of contents for the website of the Acme Content Company. The example does not contain anything other than a very simple table of contents and the mcf representation is therefore very similar to a nested HTML list.

2. The second example introduces Acme Company and its webmaster as units and also provides a lot more interesting information about the pages on the site.
3. The third example illustrates several concepts, such as namespaces, structured values and inheritance.

4.2 Example 1

```
<xml-mcf>

 <!-- BasicVocab defines some basic vocabulary that can
       be used to describe the structure of web sites. -->
 <MFC-REF XML-LINK="SIMPLE" ROLE="XML-MCF-BLOCK"
       href="http://www.standards.org/BasicVocab.mcf"/>

 <TableOfContents>
     <description>Acme Content Company Website Table of Contents</description

          <Subject>
              <name>Living Desert</name>
              <description>Wild Life Pictures taken in the Sahara</description

              <Page id="http://www.acc.com/scorpions.html">
                   <description>Scorpions in the sun</description>
                   </Page>
              <Page id="http://www.acc.com/Cactus.html">
                   <description>Photographs of a lone cactus</description>
                   </Page>
              </Subject>

          <Subject>
              <description>Dangerous Creatures</description>
              <Subject>
                   <description>Dangerous Creatures in Africa</description>
                   <parent unit="http://www.acc.com/scorpions.html"
                             inverse="true" />
                   </Subject>
              <Subject>
                   <description>Dangerous Creatures in South America</descripti
                   <Page id="http://www.acc.com/anaconda.html">
                        <description>Pictures of Anacondas</description>
                        </Page>
                   <Page id="http://www.acc.com/NinjaPenguins.html">
                        <description>The Mythical Ninja Penguins</description>
                        </Page>
                   </Subject>
              </Subject>
     </TableOfContents>
</xml-mcf>
```

The above example corresponds to the following nested list.

- Living Desert
  - Scorpions in the sun
  - Photographs of a lone cactus
- Dangerous Creatures
  - Dangerous Creatures in Africa.
    - Scorpions in the sun
  - Dangerous Creatures in South America
    - Pictures of Anacondas

- The Mythical Ninja Penguins

4.3 Example 2

In this example, we repeat most of we had in the previous example, and in addition, for each of the pages, we specify information like the size of the page, update schedule and who the author is. To help with this, we also introduce the Acme Company and the webmaster as units.

Please note that this structure itself is a little more complex (and cannot be represented using simple html lists) than that in the previous example.

```
<xml-mcf>

 <!--- BasicVocab defines some basic vocabulary that can
       be used to describe the structure of web sites. --->
 <MFC-REF XML-LINK="SIMPLE" ROLE="XML-MCF-BLOCK"
      href="http://www.standards.org/BasicVocab.mcf"/>


 <WebSite id="AcmeContentCompanyWebsite">
     <name>ACME Content Company Web Site</name>
     <siteHomePage unit="http://www.acc.com/"/>
     <helpPage unit="http://www.acc.com/help.html"/>
     <lastRevisionDate>today</lastRevisionDate>
     <toc unit="acctoc"/>
     <contactAgent unit="jb@acc.com"/>
     <objectIcon unit="http://www.acc.com/ACCLogo.jpg"/>
     </WebSite>

 <Person id="jb@acc.com">
     <name>John Brown</name>
     <description>John Brown, who amongst other things, takes care of
       the ACME web site</description>
     <contactInformation>415-937-2607</contactInformation>
     <email>jb@acc.com</email>
     <homePage unit="/people/jb.html"/>
     <employeeOf unit="AcmeContentCompany"/>
     </Person>

 <Organization id="AcmeContentCompany">
     <name>The Acme Content Company</name>
     <homePage unit="http://www.acme.com"/>
     </Organization>

 <TableOfContents id="acctoc">
     <description>Acme Content Company Website Table of Contents</description

     <Subject id="LivingDesert">
         <name>Living Desert</name>
         <description>Wild Life Pictures taken in the Sahara</description>
         <nextUpdateTime>June 1 1997</nextUpdateTime>
         <Page id="http://www.acc.com/scorpions.html">
             <description>Scorpions in the sun</description>
             <authorIndividual unit="jb@acc.com"/>
             <copyright unit="copyright.html"/>
                 <toc                unit="acctoc"/>
                 <authorOrganization unit="AcmeContentCompany"/>
             <size>2000</size>
             <loadSize>35000</loadSize>
```

```
            </Page>
        <Page id="http://www.acc.com/cobra.html">
            <description>Photographs of a lone cobra</description>
            <authorIndividual unit="jb@acc.com"/>
            <copyright        unit="copyright.html"/>
                <toc                  unit="acctoc"/>
                <authorOrganization unit="AcmeContentCompany"/>
            </Page>
        </Subject>

    <Subject>
        <description>Dangerous Creatures</description>
        <subject unit="http://mcf.yahoo.com/mcf/Recreation/Animals.mcf"/>
        <Subject>
            <description>Dangerous Creatures in Africa</description>
            <parent unit="LivingDesert" inverse="true"/>
            <!--- we are incorporating the living desert sub-tree under
                  here as well --->
            </Subject>

        <Subject>
            <description>Dangerous Creatures in South America</description>
            <Page id="http://www.acc.com/anaconda.html">
                <description>Pictures of Anacondas</description>
                <authorIndividual unit="jb@acc.com"/>
                <copyright        unit="copyright.html"/>
                    <toc                    unit="acctoc"/>
                    <authorOrganization unit="AcmeContentCompany"/>
                <includesContent unit="/images/anaconda.jpg"/>
                </Page>
            <Page id="http://www.acc.com/NinjaPenguins.html">
                <description>The Mythical Ninja Penguins</description>
                <authorIndividual unit="jb@acc.com"/>
                <copyright unit="copyright.html"/>
                    <toc                    unit="acctoc"/>
                    <authorOrganization unit="AcmeContentCompany"/>
                </Page>
            </Subject>
        </Subject>

    <Page id="copyright.html">
        <description>Copyright and Other Legal Notices</description>
        <authorOrganization unit="AcmeContentCompany"/>
        <contentUpdateSchedule unit="NeverUpdated"/>
        <language unit="USEnglish"/>
        </Page>

</TableOfContents>

</xml-mcf>
```

4.4 Example Three

In this example, we introduce several advanced features. Specifically, we illustrate,

- Schema additions (i.e., new categories and property types) made by Acme
- The use of namespaces
- The use of structured values
- Inheritance of certain properties that are true for a lot of the Acme pages.

We first provide the schema additions and then get into the description of the site.

4.4.1 Schema Extensions for Acme

The following describes the schema extensions made by the Acme Content Company that are available from http://www.acme.com/AcmeVocab.mcf This is a very small extension, but it illustrates the concept of how MCF can be used to extend itself:

```
<xml-mcf>
<!-- The contents of the MCF block that appear at
     http://www.acme.com/AcmeVocab.mcf -->

<MFC-REF XML-LINK="SIMPLE" ROLE="XML-MCF-BLOCK"
      href="http://www.standards.org/BasicVocab.mcf"/>


  <!---
     we have declared a new property called accDeptOfPage which applies
     to web pages and whose entry is an ACCDepartment. We have also said
     that there may be at most one department responsible for each page and
     that the department is also the contactAgent for the page
   --->

 <FunctionalPropertyType id="deptOfPage">
     <description>
  Every page has a department associated with it (at ACC). This property is
  used to specify the ACC department associated with the page.</description>
      <domain unit="AcmePage"/>
      <name>deptOfPage</name>
      <range unit="Department"/>
      <superProperty unit="contactAgent"/>
      </FunctionalPropertyType>
 <Category id="Department">
     <name>Department</name>
     <superType unit="Organization"/>
     <description>Departments in the Acme Content Company</description>
     </Category>

 <FunctionalPropertyType id="departmentNumber">
     <name>departmentNumber</name>
     <description>
 The ACC department number associated with an ACC department</description>
     <domain unit="Department"/>
     <range unit="Integer"/>
     </FunctionalPropertyType>

 <Category id="AcmePage">
     <name>AcmePage</name>
     <superType unit="Page"/>
     <description>All the acme web pages</description>
     <inherits unit="copyright.html" propertytype="copyright"/>
     <!--- this is equivalent to adding
           <copyright unit="copyright.html"/>
             as part of every unit whose typeOf is AcmePage --->
     <inherits unit="jb@acc.com" propertytype="authorIndividual"/>
     <inherits unit="AcmeContentCompany" propertytype="authorOrganization"/>
     <inherits unit="acctoc" propertyType="toc"/>
     <inherits unit="help.html" propertytype="helpPage"/>
     <inherits propertytype="cost">$ 0</inherits>
```

```
    <inherits propertytype="cost">$ 0</inherits>
    <!--- this is equivalent to adding
          <cost>$ 0</cost>
          as part of every unit whose typeOf is AcmePage --->
    </Category>
</xml-mcf>
```

Now the actual description of the web site that uses the above schema extensions. (The base url of the above mcf block and the following mcf block are the same. If they were not, references in the above block to content units defined in the following block might need to be adjusted.)

```
<xml-mcf>

 <MFC-REF XML-LINK="SIMPLE" ROLE="XML-MCF-BLOCK"
      href="http://www.standards.org/BasicVocab.mcf"/>
<!-- include the previously defined schema so that it is available here --->
 <MFC-REF XML-LINK="SIMPLE" ROLE="XML-MCF-BLOCK"
      href="http://www.acc.com/accExtensions.mcf" prefix="acme"/>

 <WebSite id="AcmeContentCompanyWebsite">
     <name>ACME Content Company Web Site</name>
     <siteHomePage unit="http://www.acc.com/"/>
     <helpPage unit="http://www.acc.com/help.html"/>
     <lastRevisionDate>today</lastRevisionDate>
     <toc unit="acctoc"/>
     <contactAgent unit="jb@acc.com"/>
     <objectIcon unit="http://www.acc.com/ACCLogo.jpg"/>
     </WebSite>

 <Person id="jb@acc.com">
     <name>John Brown</name>
     <description>John Brown, who amongst other things, takes care of
       the ACME web site</description>
     <contactInformation>415-937-2607</contactInformation>
     <email>jb@acc.com</email>
     <homePage unit="/people/jb.html"/>
     <employeeOf unit="AcmeContentCompany"/>
     </Person>

 <Organization id="AcmeContentCompany">
     <name>The Acme Content Company</name>
     <homePage unit="http://www.acme.com"/>
     <contactInformation>
         <address>
             <streetAddress>17 Loop Drive.</streetAddress>
             <cityAddress>Alto Palo</cityAddress>
             <stateAddress>CA</stateAddress>
             <zip>95014</zip>
             </address>
         <phoneNumber>
             <areaCode>415</areaCode>
             <phoneNumberBody>965-1279</phoneNumberBody>
             </phoneNumber>
        </contactInformation>
      </Organization>

 <acme:Department id="acc.com/accEMarketingDept.mcf">
      <departmentNumber value="32"/>
     </acme:Department>
```

```
<!--- note that we don't have to specify the author, etc. on the
      pages in this example. All of that is inherited by virtue of the
      pages being AcmePage --->

<TableOfContents id="acctoc">
    <description>Acme Content Company Website Table of Contents</description

    <Subject id="LivingDesert">
        <name>Living Desert</name>
        <description>Wild Life Pictures taken in the Sahara</description>
        <nextUpdateTime>June 1 1997</nextUpdateTime>
        <AcmePage id="http://www.acc.com/scorpions.html">
            <description>Scorpions in the sun</description>
            <size>2000</size>
                <loadSize>35000</loadSize>
        </AcmePage>
        <AcmePage id="http://www.acc.com/cobra.html">
            <description>Photographs of a lone cobra</description>
            </AcmePage>
        </Subject>

    <Subject>
        <description>Dangerous Creatures</description>
        <subject unit="http://mcf.yahoo.com/mcf/Recreation/Animals.mcf"/>
        <Subject>
            <description>Dangerous Creatures in Africa</description>
            <parent unit="LivingDesert" inverse="true"/>
            <!--- we are incorporating the living desert sub-tree under
                  here as well --->

            <Subject>
                <description>Dangerous Creatures in South America</descripti
                <AcmePage id="http://www.acc.com/anaconda.html">
                    <description>Pictures of Anacondas</description>
                    <acme:deptOfPage unit="acc.com/accEMarketingDept.mcf"/>
                    <includesContent unit="/images/anaconda.jpg"/>
                    </AcmePage>
                <AcmePage id="http://www.acc.com/NinjaPenguins.html">
                    <description>The Mythical Ninja Penguins</description>
                    </AcmePage>
                </Subject>
            </Subject>
        </Subject>

    <Page id="copyright.html">
        <description>Copyright and Other Legal Notices</description>
        <authorOrganization unit="AcmeContentCompany"/>
        <contentUpdateSchedule unit="NeverUpdated"/>
        <language unit="USEnglish"/>
        </Page>
</TableOfContents>


</xml-mcf>
```

4.4.2 Structured Values

In the previous example, we had,

```
<Organization id="AcmeContentCompany">
        ...
        <contactInformation>
            <address>
                <streetAddress>17 Loop Drive.</streetAddress>
                <cityAddress>Alto Palo</cityAddress>
                <stateAddress>CA</stateAddress>
                <zip>95014</zip>
                </address>
            <phoneNumber>
                <areaCode>415</areaCode>
                <phoneNumberBody>965-1279</phoneNumberBody>
                </phoneNumber>
            </contactInformation>
        </Organization>
```

This is equivalent to,

```
<Organization id="AcmeContentCompany">
        ...
        <contactInformation>17 Loop Drive. Alto Palo CA 415 965-1279</contact
        <address>17 Loop Drive. Alto Palo CA</address>
        <streetAddress>17 Loop Drive.</streetAddress>
        <cityAddress>Alto Palo</cityAddress>
        <stateAddress>CA</stateAddress>
        <phoneNumber>415 965-1279</phoneNumber>
        <areaCode>415</areaCode>
        <phoneNumberBody>965-1729</phoneNumberBody>
</Organization>
```

I.e., the value of the enclosing property (such as contactInformation) is the concatenation of the included values. The interior properties (such as address or streetAddress), just like the exterior properties, apply to the same container.

Appendices

A. Standard Vocabulary

In addition to the basic bootstrapping terms *typeOf*, Category, etc.) specified earlier, in order to promote interoperability, we also propose some standard vocabulary that can be used for purposes of describing the kinds of content typically found on the web.

Such standard schemata are very important, but are separate from the data model and the transfer syntax. The purpose of this section of the proposal is to initiate a discussion. There is significant work to do in this area, but it should be started now.

Though the following can easily be specified in MCF itself, for purposes of readability, we provide the following description in English. The MCF specification will however be made available for authors.

An author can use this vocabulary as the schema for their MCF (by using XML-transclusion) and make further modifications and additions to it as they need.

A.1 Categories

As a convention, Categories are in the singular. So, the category of all people is called *Person* and of all organizations

is called *Organization*.

Also, even though MCF is case insensitive, for purposes of human readability, as a convention, categories start with a capital letter and properties start with a lower case letter.

The name and identifier for all of the following are the same.

`Content`
: Includes everything from websites and web pages to legacy databases and file folders. Its superType is Unit.

`ContentContainer`
: A collection of information. Includes subject categories, file folders, channels, etc. Its superType is Content. There are no constraints on the items belonging to a container. The items in a container could themselves be containers. The relation between an item belonging to a container and the container is just parent (though we might want to eventually introduce a more specialized relation.) The distinction between a container and non-container is one of convenience. There will be cases where we want to consider a single page as a container and in other cases, we might want to consider the same page as an atomic entity. The flexibility of MCF allows us this freedom.

`Subject`
: The category of subjects. An example is the Arts category in Yahoo! or the portion of the Developer portion of the Netscape Website. Its superType is ContentContainer.

`WebSite`
: A web site. Its superType is ContentContainer.

`Page`
: A document. Could be a WordPerfect document on a PC or a web page or even a FileMaker database. Its superType is Content.

`Agent`
: The concept of an Agent is a general one intended to cover people, robots, organizations, etc. Its superType is Unit.

`Organization`
: Examples include Apple Computer, United States and the Peace Corps. Organizations are mutually disjoint with people. Its superType is Agent.

`Person`
: The category of people. Its superType is Agent.

`TableOfContents`
: The table of contents for any Content (could be for a web site, page, ...) Its superType is Content.

`NaturalLanguage`
: Examples include English, French, etc. Its superType (for now) is Unit.

`Schedule`
: This category is used to specify information like the periodicity with which content is updated, when it should be pulled down, etc. The range includes both simple instances like Hourly or Daily to instances with intermediate complexity like daily at eight am to more complex instances (such as that proposed by CDF) like hourly between eight am and six pm on weekdays...

A.2 Property Types

A.2.1 Property Types used to describe Agents

There has been much work in standardizing vocabularies for describing agents, most notably vcard, and we hope to adopt those standards as applicable. In addition, we should also provide standard property types for describing the location, hobbies, etc. of agents.

`emailAddress`
: A string representing the email address of an agent.

`homePage`

The url of the home page(s) of an agent.

`contactInformation`

A string representing how the person can be contacted.

A.2.2 property types used to describe Content

Existing standards that these draw from (and will rely upon even more in the future) include the Dublin Core, Z39.50 and of course, the rich body of work in Library Science.

A.2.2.1 Authorship Related property types

`authorIndividual`

The individual person(s) who is(are) the authors of the content object. The entries are not names of the authors but references to objects corresponding to the authors. The name, email address, etc. of the author can be specified on that object.

`authorOrganization`

The organization which is the author of the content object.

`author`

The generalization of the previous 2 property typess. The is a superPropertyType of both. of them.

`editor`

The agent that is the editor of the content object.

`publisher`

The agent that is the publisher of the content object.

`contactAgent`

The agent who is the "contact" for that piece of content. Typically the person behind "webmaster@xyz.com".

`copyright`

The copyright declarations. The range is page addressing the copyright and other legal issues.

A.2.2.2 property types related to the size of the object

`size`

The size of a content object in bytes. Represented using an integer. This is the size of the object alone and does not represent the size of its inclusions (like in-line images).

`loadSize`

The total number of bytes, including inline images, plugins, etc. of a content object.

A.2.2.3 Temporal property types of the content

Some more temporal property types appear under Schedules.

`publicationDate`

The date on which a content object was first published.

`lastRevisionDate`

The date on which the content object was last modified.

`expires`

The date until the information in this content object is valid.

`contentUpdateSchedule`

The frequency with which this is typically updated. The range is a Schedule (which includes Hourly, Daily, etc. and also more complex Schedules.)

`versionNumber`

The version number of this content object or subject category. A string.

`contentDownloadSchedule`

This is to be used if the content is to be proactively downloaded to the users computer. It specifies the download schedule and the entry is a Schedule.

`nextUpdateTime`

The next time that this piece of content is scheduled to be updated.

`nextDownloadTime`

This is also to be used if the content is to be proactively downloaded to the users computer. It specifies the next time this piece of content should redownloaded. More often than not, this will suffice in lieu of a full blown schedule and will default to the nextUpdateTime.

A.2.2.4 property types about the content itself

`subject`

The subject categories that this content object falls under. parent is a superProperty of subject. Using this, an author could for example suggest that his/her page belongs to a certain Yahoo! subject category.

`language`

The language(s) (typically a natural language such as English or French) in which the content is primarily encoded.

`toc`

One or more tables of contents of which this content object is a part.

`siteHomePage`

The home page for the site of which this content object is a part.

`helpPage`

The page at which help can be found regarding this content object.

`linksTo`

The content objects that a content object has hyperlinks to. parent is a superProperty of linksTo.

`includesContent`

To be used when one content object includes another (such as an HTML page including an image or a poem). This is useful when we want to distinctly identify a certain piece of a page, such as a table, as a first class unit and specify the relation between the enclosing page and table.

`contentMimeType`

The MIME type of the content.

`contentPartMimeTypes`

A convenience predicate for specifying the mime types of all the included content.

`superTopic`

A relational between two subject categories such as Yahoo Arts and Yahoo Arts Museums which states that the later is a more specific subject category of the former. parent is a superPropertyType of superTopic.

`objectIcon`

An icon that can be used to represent the object. The value is typically the object corresponding to a GIF or JPEG, but could also be a platform specific encoding. Preferably, it will be one object with several different encodings being available.

A.2.2.5 property types about content access

`location`

One or more URIs at from which object content may be obtained.

`contentMirror`

Mirror uris for this content object. Mirrors are assumed to be secondary sources of the content, which might potentially be stale. The distinction between mirrors and location is subtle at best.

`contentAvailabilityStatus`

This Property can be used to specify information like whether the server is down, the last time the content was accessible, etc. This meta-content is typically furnished not by the content provider himself, but by indexers like Yahoo!

A.2.2.6 Other property types about content

`accessMode`

This is used to specify whether the content is to be accessed via the traditional Web pull mechism, via email (e.g., InBox Direct), via channels, etc.

`contentRating`

The intent of this Property is to contain the information that would be contained in a PICs-like rating. The range is Rating. The schemas for Ratings is beyond the scope of this document.

`contentCost`

The cost of this content. The range is a Cost, which could be as simple as "5 US Dollars" or something much more complex. The more complex specification is beyond the scope of this proposal.

A.2.3 property types related to schedules.

`scheduleStartDate`

This is the day upon which the schedule will start to apply.

`scheduleEndDate`

This is the day upon which the schedule expires and no longer applies.

`scheduleIntervalTime`

The interval of time that the schedule should repeat over.

`scheduleEarliestTime`

Earliest time during the schedule interval that the schedule applies to.

B. Acknowledgements

A very large number of people have contributed to the material in this proposal. It draws heavily from the knowledge representation work in AI. It owes a lot to the MCF project at Apple and we would like to thank the folks who made that happen, including Jed Harris, Alan Kay, Don Norman and Larry Tesler. We would also like to thank Edwin Aoki, Tim Craycroft, Tim Hickman, Phil Karlton, Mike McCue and Tom Paquin of Netscape for the comments and feedback on this draft. External feedback from Jon Bosak and Mark Walter was also of great help.

W3C

XML-Data

W3C Note 05 Jan 1998

This version:
http://www.w3.org/TR/1998/NOTE-XML-data-0105
Latest version:
http://www.w3.org/TR/1998/NOTE-XML-data
Authors:
Andrew Layman, Microsoft Corporation
Edward Jung, Microsoft Corporation
Eve Maler, ArborText
Henry S. Thompson, University of Edinburgh
Jean Paoli, Microsoft Corporation
John Tigue, DataChannel
Norbert H. Mikula, DataChannel
Steve De Rose, Inso Corporation

Status of this Document

This document is a NOTE made available by the World Wide Web Consortium for discussion only. This indicates no endorsement of its content, nor that the Consortium has, is, or will be allocating any resources to the issues addressed by the NOTE. A list of current NOTEs can be found at: http://www.w3.org/TR/.

This document is a submission to the W3C. Please see Acknowledged W3C Submissions regarding its disposition.

Contents:

Acknowledgements

We thank Paul Grosso(ArborText), Sharon Adler (Inso Corporation), Anders Berglund (Inso Corporation), François Chahuneau (AIS/Berger-Levrault)for their help and contributions to this proposal.

Introduction

Schemas define the characteristics of classes of objects. This paper describes an XML vocabulary for schemas, that is, for defining and documenting object classes. It can be used for classes which as strictly syntactic (for example, XML) or those which indicate concepts and relations among concepts (as used in relational databases, KR graphs and RDF). The former are called "syntactic schemas;" the latter "conceptual schemas."

For example, an XML document might contain a "book" element which lexically contains an "author" element and a "title" element. An XML-Data schema can describe such syntax. However, in another context, we may simply want to represent more abstractly that books have titles and authors, irrespective of any syntax. XML-Data schemas can describe such conceptual relationships. Further, the information about books, titles and authors might be stored in a relational database, in which XML-Data schemas describe row types and key relationships.

One immediate implication of the ideas in this paper is that XML document types can now be described using XML itself, rather than DTD systax. Another is that XML-Data schemas provide a common vocabulary for ideas which overlap between syntactic, database and conceptual schemas. All features can be used together as appropriate.

Schemas are composed principally of declarations for:

- Concepts
- Classes of objects
  - Class hierarchies
  - Properties
  - Constraints
  - Relationships
  - Indicated by primary key to foreign key matching
  - Indicated by URI
- XML DTD Grammars and Compatibility
  - grammatical rules governing the valid nesting of the elements and attributes
  - attributes of elements
  - internal and external entities, represented by intEntityDecl and extEntityDecl

    - notations, represented by notationDcl
- Datatypes giving parsing rules and implementation formats.
- Mapping rules allowing abbreviated grammars to map to a conceptual data model.

The Schema Element Type

All schema declarations are contained within a schema element, like this:

```
<?XML version='1.0' ?>
<?xml:namespace name="urn:uuid:BDC6E3F0-6DA3-11d1-A2A3-00AA00C14882/" as="s
<s:schema id='ExampleSchema'>
  <!-- schema goes here. -->
</s:schema>
```

The namespace of the vocabulary described in this document is named "urn:uuid:BDC6E3F0-6DA3-11d1-A2A3-00AA00C14882/".

The *ElementType* Declaration

The heart of an XML-Data schema is the *elementType* declaration, which defines a class of objects (or "type of element" in XML terminology). The *id* attribute serves a dual role of identifying the definition, and also naming the specific class.

```
<elementType id="author"/>
```

Within an elementType, the *description* subelement may be used to provide a human-readable description of the element's purpose.

```
<elementType id="author">
  <description>The person, natural or otherwise, who wrote the book.</descr
</elementType >
```

Properties and Content Models

Subelements within *elementType* define characteristics of the class's members. An XML "content model" is a description of the contents that may validly appear within a particular element type in a document instance.

```
<elementType id="author">
    <string/>
</elementType>

<elementType id="Book">
    <element type="#author" occurs="ONEORMORE"/>
</elementType>
```

The example above defines two elements, author and book, and says that a book has one or more authors. The author element may contain a string of character data (but no other elements). For example, the following is valid:

```
<Book>
```

```
    <author>Henry Ford</author>
    <author>Samuel Crowther</author>
</Book>
```

Within an elementType, various specialized subelements (element, group, any, empty, string etc.) indicate which subelements (properties) are allowed/required. Ordinarily, these imply not only the cardinality of the subelements but also their sequence. (We discuss a means to relax sequence later.)

Element

Element indicates the containment of a single element type (property). Each *element* contains an *href* attribute referencing another *elementType*, thereby including it in the content model syntactically, or declaring it to be a property of the object class conceptually. The element may be required or optional, and may occur multiple times, as indicated by its *occurs* attribute having one of the four values "REQUIRED", "OPTIONAL", "ZEROORMORE" or "ONEORMORE". It has a default of "REQUIRED".

```
<elementType id="Book">
    <element type="#title" occurs="OPTIONAL"/>
    <element type="#author" occurs="ONEORMORE"/>
</elementType>
```

The example above describes a book element type. Here, each instance of a book *may* contain a title, and *must* contain one or more authors.

```
<Book>
    <author>Henry Ford</author>
    <author>Samuel Crowther</author>
    <title>My Life and Work</title>
</Book>
```

When we discuss type hierarchies, later, we will see that an element type may have subtypes. If so, inclusion of an element type in a content model permits elements of that type directly and all its subtypes.

Empty, Any, String, and Mixed Content

Empty and *any* content are expressed using predefined elements *empty* and *any*. (*Empty* may be omitted.) *String* means any character string not containing elements, known as "PCDATA" in XML. *Any* signals that any mixture of subelements is legal, but no free characters. *Mixed* content (a mixture of parsed character data and one or more elements) is identified by a *mixed* element, whose content identifies the element types allowed in addition to parsed character data. When the content model is mixed, any number of the listed elements are allowed, in any order.

```
<?XML version='1.0' ?>
<?xml:namespace name="urn:uuid:BDC6E3F0-6DA3-11d1-A2A3-00AA00C14882/" as="s
<s:schema>

    <elementType id="name">
        <string/>
    </elementType>

    <elementType id="Person">
        <any/>
    </elementType>

    <elementType id="author">
        <string/>
    </elementType>
```

```
    <elementType id="titlePart">
        <string/>
    </elementType>

    <elementType id="title">
        <mixed><element type="#titlePart"/></mixed>
    </elementType>

    <elementType id="Book">
        <element type="#title" occurs="OPTIONAL"/>
        <element type="#author" occurs="ONEORMORE"/>
    </elementType>

</s:schema>

...

<Book>
    <author>Henry Ford</author>
    <author>Samuel Crowther</author>
    <title>My Life and<titlePart>Work</titlePart></title>
</Book>
```

Here, *book* is defined to have an optional *title* and one or more *authors*. The *name* element has content model of *any*, meaning that free text is not allowed, but any arrangement of subelements is valid. The *content model* of *title* is *mixed*, allowing a free intermixture of characters and any number of *titleParts*. The author, *name* and *titleParts* elements have a *content model* of *string*.

Group

Group indicates a set or sequence of elements, allowing alternatives or ordering among the elements by use of the groupOrder attribute. The group as a whole is treated similarly to an element.

```
<elementType id="Book">
    <element type="#title"/>
    <element type="#author" occurs="ONEORMORE"/>
    <group occurs="OPTIONAL">
        <element type="#preface"/>
        <element type="#introduction"/>
    </group>
</elementType>
```

In the above example, if a preface or introduction appears, both must, with the preface preceding the introduction. Each of the following is valid:

```
<Book>
    <author>Henry Ford</author>
</Book>

<Book>
    <author>Henry Ford</author>
    <preface>Prefatory text</preface>
    <introduction>This is a swell book.</introduction>
</Book>
```

Sometimes a schema designer wants to relax the ordering restrictions among elements, allowing them to appear in any order. This is indicated by setting the groupOrder attribute to "AND":

```
<elementType id="Book">
    <element type="#title"/>
    <element type="#author" occurs="ONEORMORE"/>
    <group groupOrder="AND" occurs="OPTIONAL">
        <element type="#preface"/>
        <element type="#introduction"/>
    </group>
</elementType>
```

Now the following is also valid:

```
<Book>
    <author>Henry Ford</author>
    <introduction>This is a swell book.</introduction>
    <preface>Prefatory text</preface>
</Book>
```

Finally, a schema can indicate that any one of a list of elements (or groups) is needed. For example, either a preface *or* an introduction. The groupOrder attribute value "OR" signals this.

```
<elementType id="Book">
    <element type="#title"/>
    <element type="#author" occurs="ONEORMORE"/>
    <group groupOrder="OR">
        <element type="#preface"/>
        <element type="#introduction"/>
    </group>
</elementType>
```

Now each of the following is valid:

```
<Book>
    <author>Henry Ford</author>
    <preface>Prefatory text</preface>
</Book>

<Book>
    <author>Henry Ford</author>
    <introduction>This is a swell book.</introduction>
</Book>
```

Open and Closed Content Models

XML typically does not allow an element to contain content unless that content was listed in the model. This is useful in some cases, but overly in others in which we would like the listed content model to govern the cardinality and other aspects of whichever subelements are explicitly named, while allowing that other subelements can appear in instances as well.

The distinction is effected by the *content* attribute taking the values "OPEN" and "CLOSED." The default is "OPEN" meaning that all element types not explicitly listed are valid, without order restrictions. (This idea has a close relation to the Java concept of a final class.)

For example, the following instance data for a book, including the unmentioned element *copyrightDate* would be valid given the content models declared so far, because they have all been *open*.

```
<Book>
    <author>Henry Ford</author>
    <author>Samuel Crowther</author>
```

```
    <title>My Life and Work</title>
    <copyrightDate>1922</copyrightDate>
</Book>
```

However, had the content model been declared closed, as follows, the *copyrightDate* element would be invalid.

```
<elementType id="Book" content="CLOSED">
    <element type="#title"/>
    <element type="#author" occurs="ONEORMORE"/>
    <group groupOrder="SEQ" occurs="OPTIONAL">
        <element type="#preface"/>
        <element type="#introduction" occurs="REQUIRED"/>
    </group>
</elementType>
```

A closed content model does not allow instances to contain any elements or attributes beyond those explicitly listed in the elementType declaration.

Default Values

An element with occurs of REQUIRED or OPTIONAL (but not ONEORMORE or ZEROORMORE) can have a default value specified.

```
<elementType id="Book">
    <element type="#title"/>
    <element type="#author" occurs="ONEORMORE"/>
    <element type="#ageGrp" occurs="OPTIONAL">
        <default>adult</default>
    </element>
</elementType>
```

The default value is implied for all element *instances* in which it is syntactically omitted.

To indicate that the default value is the only allowed value, the *presence* attribute is set to "FIXED".

```
<elementType id="Book">
    <element type="#title"/>
    <element type="#author" occurs="ONEORMORE"/>
    <element type="#ageGrp" occurs="OPTIONAL" presence="FIXED">
        <default>ADULT</default>
    </element>
</elementType>
```

Presence has values of "IMPLIED," "SPECIFIED," "REQUIRED," and "FIXED" with the same meanings as defined in XML DTD.

Aliases and Correlatives

ElementTypes can be know be different names in different languages or domains. The equivalence of several names is effected by the sameAs attribute, as in

```
<elementTypeEquivalent id="livre" type="#Book"/>
```

```
<elementTypeEquivalent id="auteur" type="#author"/>
```

Elements are used to represent both primary object types (nouns) and also properties, relations and so forth. Relations are often known by two names, each reflecting one direction of the relationship. For example, husband and wife, above and below, earlier and later, etc. The *correlative* element identifies such a pairing.

```
<elementType id= "author">
    <string/>
</elementType>

<elementType id= "wrote">
    <correlative type="#author" />
    <string/>
</elementType>
```

This indicates that "wrote" is another name for the "author" relation, but from the perspective of the person, not the book. That is, the two fragments below express the same fact:

```
<Person>
    <name>Henry Ford</name>
</Person>

<Book>
    <title>My Life and Work</title>
    <author>Henry Ford</author>
</Book>

...

<Person>
    <name>Henry Ford</name>
    <wrote>My Life and Work</wrote>
</Person>

<Book>
    <title>My Life and Work</title>
</Book>
```

A correlative may be defined simply to document the alternative name for the relation. However, it may also be used within a content model where it permits instances to use the alternative name. Further it may to establish constraints on the relation, indicate key relationships, etc.

Class Hierarchies

ElementTypes can be organized into categories using the *superType* attribute, as in

```
<elementType id="price">
    <string/>
</elementType>

<elementType id="ThingsI'veBoughtRecently">
    <element type="#price"/>
</elementType>

<elementType id="PencilsI'veBoughtRecently">
    <superType type="#ThingsI'veBoughtRecently"/>
```

```
    <element type="#price"/>
</elementType>

<elementType id="BooksI'veBoughtRecently">
    <superType type="#ThingsI'veBoughtRecently"/>
    <element type="#price"/>
</elementType>
```

This simply indicates that, in some fashion, *PencilsI'veBoughtRecently* and *BooksI'veBoughtRecently* are subsets of *ThingsI'veBoughtRecently*. It implies that every valid instance of the subset is a valid instance of the superset. The superset type must have an *open* content model.

There are restrictions that should be followed, based on the principle that all instances of the species (subtype) must be instances of the genus (supertype):

- The genus type must have content="OPEN".
- It must have either no groups or only groups with groupOrder="AND" (that is, no order constraints).
- You can add new elements and attributes.
- Occurs cardinality can be decreased but not increased.
- Ranges and other constraints are cummulative, that is, all apply (though the exact effect of this depends on the semantics of the constraint type).
- Default values can be made FIXED defaults.

To indicate that the content model of the subset should inherit the content model of a superset, we use a particular kind of superType called "genus" of which only one is allowed per ElementType. This copies the content model of the referenced element type and permits addition of new elements to it. Further, sub-elements occurring in the superset type, if declared again, are replaced by the newer declarations.

```
<elementType id="Book">
    <element type="#title"/>
    <element type="#author" occurs="ONEORMORE"/>
</elementType>

<elementType id="BooksI'veBoughtRecently">
    <genus type="#Book"/>
    <superType type="#ThingsI'veBoughtRecently"/>
    <element type="#price"/>
</elementType>
```

The above has the same effect as

```
<elementType id="Book">
    <element type="#title"/>
    <element type="#author" occurs="ONEORMORE"/>
</elementType>

<elementType id="BooksI'veBoughtRecently">
    <superType type="#Book"/>
    <superType type="#ThingsI'veBoughtRecently"/>
    <element type="#title"/>
    <element type="#author" occurs="ONEORMORE"/>
    <element type="#price"/>
</elementType>
```

Elements which are References

ElementTypes and the content model elements defined so far are sufficient to declare a tree structure of elements. However, some elements such as "author" are *not only* usable on their own, they also act as references to other elements. For example, "Henry Ford" is the value of the *author* subelement of a *book* element. "Henry Ford" is also the value of the *name* element in a *person* element, and it can be used to connect these two.

```
<Book>
    <author>Henry Ford</author>
    <author>Samuel Crowther</author>
    <title>My Life and Work</title>
</Book>

<Person><name>Henry Ford</name></Person>

<Person><name>Samuel Crowther</name></Person>
```

In this capacity, such subelement are often referred to as *relations* when using "knowledge representation" terminology or "keys" when using database terms. (The meaning of "relation" and "key" are slightly different, but the fact which the terms recognize is the same.)

To make such references explicit in the schema, we add declarations for *keys* and *foreign keys*.

```
<elementType id="name">
    <string/>
</elementType>

<elementType id="Person">
    <element id="p1" type="#name"/>
    <key id="k1"><keyPart href="#p1"/></key>
</elementType>

<elementType id="author">
    <string/>
    <foreignKey range="#Person" key="#k1"/>
</elementType>

<elementType id="title">
    <string/>
</elementType>

<elementType id="Book">
    <element type="#title"/>
    <element type="#author" occurs="ONEORMORE"/>
</elementType>
```

The *key* element within *person* tells us that a person can be uniquely identified by his *name*. The *foreignKey* element within the *author* element definition says that the contents of an author element are a foreign key indentifying a person by *name*.

An uninformed user agent can still display the string "Henry Ford" even if it cannot determine that is supposed to be a person. A savvy agent that reads the schema can do more. It can locate the actual person.

This is the information needed for a *join* in database terminology.

This mechanism not only handles the typical way in which properties are expressed in databases, it also handles all cases in which the contents of an element are to be interpreted as strings from a restricted vocabulary, such as enumerations, XML nmtokens, etc.

```
<Book>
    <author>Henry Ford</author>
    <author>Samuel Crowther</author>
    <title>My Life and Work</title>
    <lccn>HD9710.U54 F58 1973</lccn>
    <dewey>629.2/092/4 B</dewey >
    <isbn>0405050887</isbn>
    <series>Business<series>
</Book>
```

Although not shown here, presumably *lccn*, *dewey* and *isbn* are declared in the schema to be foreign keys to corrresponding fields of catalog records. *Series* is a foreign key to a categorization of books, of which "Business" is one category.

Keys can contain URIs, as in

```
<Book>
    <author>http://SSA.gov/blab/people/Henry+Ford</author>
    <author>http://SSA.gov/blab/people/Samuel+Crowther</author>
    <title>My Life and Work</title>
</Book>
```

This is indicated in the schema by a datatype of "URI".

```
<elementType id="author">
    <string/>
    <datatype dt="uri"/>
</elementType>
```

One-to-Many Relations

Element relations are binary. That is, we never express an n-to-1 relationship directly. We do not, for example, list within *books* a single relation that somehow resolves to all the *authors*. Instead, we always write the relationship on the 1-to-n side, but allow multiple occurrances of the *subelement*, for example, allowing *books* to have multiple occurrences of *author*.

```
<Person><name>Henry Ford</name></Person>

<Person><name>Samuel Crowther</name></Person>

<Person><name>Harvey S. Firestone</name></Person>

<Book>
    <author>Henry Ford</author>
    <author>Samuel Crowther</author>
    <title>My Life and Work</title>
</Book>

<Book>
    <author>Harvey S. Firestone</author>
    <author>Samuel Crowther</author>
    <title>Men and Rubber</title>
</Book>
```

This example shows a book with several persons as author, and also a person who is author of several books. We discussed such many-to-many relations more under the topic of *correlations*.

Multipart Keys

When the foreignKey element does not have foreignKeyPart sub-elements (as it does not above) then the entirety of the element's contents (e.g. "Henry Ford") should be used as the key value. However, for multipart foreign keys, or cases where the element has several sub-elements, foreignKeyPart is used, as shown below.

```
<elementType id="firstName">
    <string/>
</elementType>

<elementType id="lastName">
    <string/>
</elementType>

<elementType id="Person">
    <element id="pp1" type="#firstName"/>
    <element id="pp2" type="#lastName"/>
    <key id="k1">
        <keyPart href="#pp1"/>
        <keyPart href="#pp2"/>
    </key>
</elementType>

<elementType id="author">
    <element id="ap1" type="#firstName"/>
    <element id="ap2" type="#lastName"/>
    <domain type="#Book"/>
    <range type="#Person"/>
    <foreignKey range="#Person" key="#k1">
        <foreignKeyPart href="#ap1"/>
        <foreignKeyPart href="#ap2"/>
    </foreignKey>
</elementType>

...

<Book>
    <title>My Life and Work</title>
    <author>
        <firstName>Henry</firstName>
        <lastName>Ford</lastName>
    </author>
</Book>
```

Attributes as References

An alternative way to express a reference is with an attribute.

```
<person id="person1"><name>Henry Ford</name></Person>

<person id="person2"><name>Samuel Crowther</name></Person>

<Book>
    <author name="Henry Ford"/>
    <author name="Samuel Crowther"/>
    <title>My Life and Work</title>
</Book>
```

This allows us to link a book to a person, through the author relation, using an attribute of the relation. This exactly parallels the construction we saw above under "multipart keys," where a *subelement* of author contained the author's name. Here, an *attribute* of author contains the name. We can express this in our schema as

```
<elementType id="author">
    <attribute name="name" id="authorname"/>
    <foreignKey range="#Person" key="#k1">
        <foreignKeyPart href="#authorname"/>
   </foreignKey>
</elementType>
```

A widely-used variant of this is to use a URI as a foreign key:

```
<Book>
    <author href="http://SSA.gov/blab/people/Henry+Ford"/>
    <author href="http://SSA.gov/blab/people/Samuel+Crowther"/>
    <title>My Life and Work</title>
</Book>
```

In this case, we are using the *href* attribute to contain a URI. This is a particular kind of foreign key, where the *range* is any possible resource, and where that resource is not identified by some combination of its properties but instead by a name-resolution service. We indicate this by using an attribute element, with dt= "URI".

```
<elementType id="author">
    <attribute name="href" id="authorhref" dt="uri"/>
</elementType>
```

Constraints & Additional Properties

Min and Max Constraints

Elements can be limited to restricted ranges of values. The *min* and *max* elements define the lower and upper bounds.

```
<elementType id="age">
    <string/>
</elementType>

<elementType id="Person">
    <element hef="#age"><min>0</min><max>131</max></element>
</elementType>
```

Such intervals are *half-open* (that is, the *min* value is in the interval, and the *max* value is the smallest value not in the interval).

This rule leads to the simplest calculation in most cases, and is unambiguous with respect to precision. In the above example, it is clear by these rules the 130.9999 is in the interval and 131 is not. However, had we said "all numbers from 0 to 130.99," in practice we would have some ambiguity regarding the status of 130.9999. Or interpretation would depend on the precision that we inferred for the original statement. The issue is particularly ambiguous for dates. (What exactly does "From December 5 to December 8" mean? The use of half-open intervals for representation does not, however, put any requirements on how processors must display intervals. For example, dates in some contexts display differently than their storage. That is, the interval <min>1997-12-05</min><max>1997-12-09</max> might be displayed as "December 5 through December 8".

In certain cases this rule for a half-open interval is impractical (for example, what letter follows "z" in the latin alphabet?) If so, use *maxInclusive*:

```
<elementType id="student">
    <element type="#grade"><min>A</min><maxInclusive>Z</maxInclusive></elem
</elementType>
```

Domain and Range Constraints

We can use the *domain* and *range* elements to add constraints to an element's use or value. The *domain* element, if present, indicates that the element may only be used as a property of certain other elements. That is, syntactically it may appear only in the content model of those other element types. It constrains the sorts of schemas that can be written with the element.

```
<elementType id="author">
    <string/>
    <domain type="#Book"/>
    <attribute name="href" dt="uri"/>
</elementType>
```

The *domain* property above permits *author* elements to be used only within elements which are either *books* or subsets of *books*. Use of domain is optional. If omitted, there is simply no restriction.

The *range* element is used with elements which are references and declares a restriction on the types of elements to which the relation may refer. Graphically, it describes the target end of a directed edge. Each *range* element references one elementType, any of which are valid. In this case, below, we have said that an *author* element must have an *href* attribute which is a URI reference to a *Person* or to an element type which is *Person* or a subset of *Person*.

```
<elementType id="author">
    <string/>
    <domain type="#Book"/>
    <attribute name="href" dt="uri" range="#Person" />
</elementType>
```

Other useful properties

Element and attribute types can have an unlimited amount of further information added to them in the schema due to the open nature of XML with namespaces.

Using Elements from Other Schemas

A schema may use elements and attributes from other schemas in content models. For example, a subelement named "http://books.org/date" could be used within a *book* element as follows:

```
<?XML version='1.0' ?>
<?xml:namespace name="urn:uuid:BDC6E3F0-6DA3-11d1-A2A3-00AA00C14882/" as="s
<s:schema>
    <elementType id="author">
        <string/>
    </elementType>

    <elementType id="title">
        <string/>
    </elementType>
```

```
    <elementType id="Book">
        <element type="#title" occurs="OPTIONAL"/>
        <element type="#author" occurs="ONEORMORE"/>
        <element href="http://books.org/date" />
    </elementType>
</s:schema>
```

This can be abbreviated by adopting the rule that namespace-qualified names may be used within the *href attribute* value of an *element* or *attribute* element.

```
<?XML version='1.0' ?>
<?xml:namespace name="urn:uuid:BDC6E3F0-6DA3-11d1-A2A3-00AA00C14882/" as="s
<?xml:namespace name=" http://books.org/" as="bk"/?>
<s:schema>
    <elementType id="author">
        <string/>
    </elementType>

    <elementType id="title">
        <string/>
    </elementType>

    <elementType id="Book">
        <element type="#title" occurs="OPTIONAL"/>
        <element type="#author" occurs="ONEORMORE"/>
        <element href="bk:date" />
    </elementType>
</s:schema>
```

XML-Specific Elements

Attributes

XML-Data schemas contain a number of facilities to match features of XML DTDs or to support certain characteristics of XML. The XML syntax allows that certain properties can be expressed in a form called "attributes." To support this, an elementType can contain attribute declarations, which are divided into attributes with enumerated or notation values, and all other kinds.

An attribute may be given a default value. Whether it is required or optional is signaled by *presence*. (Presence ordinarily defaults to IMPLIED, but if omitted and there is an explicit default, presence is set to the SPECIFIED.) See the DTD at the end of this document for syntactic details.

Attributes with enumerated (and notation) values permit a values attribute, a space-separated list of legal values. The values attribute is required when the atttype is ENUMERATION or NOTATION, else it is forbidden. In these cases, if a default is specified, it must be one of the specified values.

```
<elementType id="Book">
    <element type="#title"/>
    <element type="#author" occurs="ONEORMORE"/>
    <attribute name="copyright" />
    <attribute name="ageGrp" atttype="ENUMERATION" values="child adult" def
</elementType>
```

describes an instance such as

```
<book copyright="1922" ageGrp="adult">
    <title>My Life and Work</title>
    <author>
        <firstName>Henry</firstName>
        <lastName>Ford</lastName>
    </author>
</Book>
```

Attributes may also reference elementTypes, meaning that one may use the element type but with attribute syntax. This allows an attribute to explicitly have the same name and semantics even when used on different element types. There are of course some limits: The attribute can still occur only once in an instance, and it cannot contain other elements. However, this allows the semantics of the element type to be employed in attribute syntax.

```
<elementType id="Book">
    <attribute href="bk:title"/>
    <attribute href="bk:author"/>
    <attribute name="copyright" />
    <attribute name="ageGrp" type="ENUMERATION" values="children adult" def
</elementType>
```

describes an instance such as

```
<book  bk:author="Henry Ford"  bk:title="My Life and Work" ageGrp="adult"/>
```

The internal and external entity declaration element type: intEntityDcl and extEntityDcl

This and the next two declarations cover entities. Entities are a shorthand mechanism, similar to macros in a programming language.

```
<intEntityDcl name="LTG">
    Language Technology Group
</intEntityDcl>

<extEntityDcl name="dilbert" notation="#gif"
              systemId="http://www.ltg.ed.ac.uk/~ht/dilb.gif"/>
```

Here as elsewhere, following XML, systemId must be a URI, absolute or relative, and publicId, if present, must be a Public Identifier (as defined in ISO/IEC 9070:1991, Information technology -- SGML support facilities -- Registration procedures for public text owner identifiers). If a notation is given, it must be declared (see below) and the entity will be treated as binary, i.e., not substituted directly in place of references.

```
<notationDcl name="gif" systemId='http://who.knows.where/'/>
```

The external declarations element type: extDcls

Although we allow an external entity with declarations to be included, we recommend a different declaration for schema modularization. The extDcls declaration gives a clean mechanism for importing (fragments of) other schemas. It replaces the common SGML idiom of declaring an external parameter entity and then immediately referring to it, and has the same import, namely, that the text referred to by the combination of systemId and publicId is included in the schema in place of the extDcls element, and that replacement text is then subject to the same validity constraints

and interpretation as the rest of the schema.

Note that in many cases the desired effect may be better represented by referencing elements (and attributes) from the other schema or subclassing from them.

Datatypes

A datatype indicates that the contents of an element can be interpreted as both a string and also, more specifically, as an object that can be interpreteted more specifically as a number, date, etc. The datatype indicates that the element's contents can be parsed or interpreted to yeild an object more specific than a string.

That is, we distinguish the "type" of an element from its "datatype." The former gives the semantic meaning of an element, such as "birthday" indicating the date on which someone was born. The "datatype" represents the parser class needed to decode the element's contents into an object type more specific than "string." For example, "19541022" is the 22nd of October, 1954 in ISO 8601 date format. (That is, ISO 8601 parsing rules will decode "19541022" into a date, which can then be stored as a date rather than a string.

For example, we would like an XML author to be able to say that the contents of a "size" element is an integer, meaning that it should be parsed according to numeric parsing rules and that it can be stored in integer format. In some contexts an API can expose it as an integer rather than a string.

```
<item>
    <name>shirt</name>
    <size>8</size>
</item>
```

There are two main contexts for datatypes. First, when dealing with database APIs, such as ODBC, all elements with the same name typically contain the same type of contents. For example, all sizes contain integers or all birthdays contain dates. We will return to this case shortly.

Second, and by contrast, the type of the content may vary widely from instance to instance. The softer we make our software, the more often these flexible cases occur. For example, size could contain the integer 8, or the word "small" or even a formula for computing the size.

We expose the datatype of an element instance by use of a *dt:dt* attribute, where the value of the attribute is a URI giving the datatype. (The URI might be explicitly in URI format or might rely on the XML namespace facility for resolution.) For example, we might find a document containing something like:

```
<?namespace name="urn:uuid:C2F41010-65B3-11d1-A29F-00AA00C14882/" as="dt"?>
<?namespace name="http://zoosports.com/dt?" as="zoo"?>
<purchases>
  <item>
    <name>shirt</name>
    <size dt:dt="int">8</size>
  </item>
  <item>
    <name>shoes</name>
    <size>large</size>
  </item>
  <item>
```

```
    <name>suit</name>
    <size dt:dt="zoo:script">
        =(shirtsize*1.05) + 3
    </size>
  </item>
</purchases>
```

Clearly this technique works for the heterogeneous typing in the above example. It also works for the database case where all element's of the same type have the same datatype.

```
<item> <name>shirt</name> <size dt:dt="int">8</size>   </item>
<item> <name>shoes</name> <size dt:dt="int">6</size>   </item>
<item> <name>suit</name>  <size dt:dt="int">12</size>  </item>
```

As written above, this is inefficient. Fortunately, XML allows us in schemasto put attributes with default or fixed values, so we could say once that all *size* elements have a datatype with value "int". Having done so, our our instance just looks like:

```
<item> <name>shirt</name> <size>14</size>  </item>
<item> <name>shoes</name> <size>6</size>   </item>
<item> <name>suit</name>  <size>16</size>  </item>
```

In a DTD, we can set a *fixed* attribute value, so that all *size* elements have datatype "int" or we can set it as a *default* attribute value so that it is an integer except where explicitly noted otherwise.

```
<item> <name>shirt</name> <size>14</size>  </item>
<item> <name>shoes</name> <size dt:dt="string">large</size>   </item>
<item> <name>suit</name>  <size>16</size>  </item>
```

XML DTDs today allow such attributes. For example, a DTD can say that all *shirt* elements have *integer datatype* by the following:

```
<!ELEMENT size PCDATA >
<!ATTLIST size   dt:dt "int" #FIXED >
```

XML-Data schemas allow the equivalent, though with specialized syntax:

```
<elementType id="size" >
    <datatype dt="int" />
</elementType>
```

Elements use *datatype* subelements to give the datatype so that an optional *presence* attribute of the datatype element can indicate whether the datatype is *fixed* or merely a *default*. Attributes can also have datatypes. Because there is no possibility of their being anything other than a fixed type, the datatype of an attribute is signalled by a dt attribute:

```
<attribute id="size" dt="int" />
```

How Typed Data is Exposed in the API

Different APIs to typed data will use the datatype attribute differently. The basic XML parser API should expose all element contents as strings regardless of any datatype attribute. (It might also contain supplementary methods to read values as more specific types such as "integer," thereby getting more efficiency.) An ODBC interface could use the datatype attribute to expose each type of element as a column, with the column's datatype determined by the element type's datatype.

Complex Data Types

If a datatype requires a complex structure for storage, or an object-based storage, this is also handled by the dt:dt attribute, where the datatype's storage format can be a structure, Java class, COM++ class, etc. For example, if an application needed to have an element stored in a "ScheduleItem" structure and using some private format, it could note this like

```
<when dt:dt="zoo:ScheduleItem">M*D1W4B19971022;100</when>
```

The datatype does not require a private format. It could also use subelements and attributes such as

```
<when dt:dt="zoo:ScheduleItem2">
    <month>*</month>
    <day>1</day>
    <week>4</week>
    <begin>19971022</begin>
    <recurs>100</recurs>
</when>
```

In the case of the graph-oriented interfaces (e.g. XML/RDF) the mapping from the XML tree to a graph should add a *wrapping node* for each non-string data type. The datatype property gives the type of that node. For example, the following two are graphically equivalent:

```
<size dt:dt="int">8</size>
<size><dt:int>8</dt:int></size>
```

Versioning of Instances

Adding an attribute to an element does not change the other attributes or pose any special versioning problems. For example, an application written to expect an instance to contain "<birthday>19541022</birthday>" is not harmed if the schema reveals that this is ISO 8601 format. Versioning within datatypes should be handled by the author's making sure that that subtypes of datatypes retain all the characteristics of the supertype.

If a down-level application is given a datatype it cannot process, it should expose the element contents as a supertype of the indicated datatype. In practice, this will usually mean that unrecognized datatypes will be the same as "dt:string". However, there are cases in which a type will be promoted, for example exposing a boolean in a byte or word rather than a bit, exposing a floating point number in a language's native format, etc.

The Datatypes Namespace

The datatype attribute "dt" is defined in the namespace named "urn:uuid:C2F41010-65B3-11d1-A29F-00AA00C14882/". (See the XML Namespaces Note at the W3C site for details of namespaces.) The full URN of the attribute is "urn:uuid:C2F41010-65B3-11d1-A29F-00AA00C14882/dt".

You will have noticed that the value of the attribute, as used in the examples above, is not lexically a full URI. For example, it reads "int" or "string" etc. Datatype attribute values are abbreviated according to the following rule: If it does not contain a colon, it is a datatype defined in the datatypes namespace "urn:uuid:C2F41010-65B3-11d1-A29F-00AA00C14882/". If it contains a colon, it is to be expanded to a full URI according to the same rules used for other names, as defined by the XML Namespaces Note. For example

```
<?namespace name="urn:uuid:C2F41010-65B3-11d1-A29F-00AA00C14882/" as="dt"?>
<?namespace name="http://zoosports.com/dt?" as="zoo"?>
<item>
    <size dt:dt="int">8</size>
    <name dt:dt="zoo:clothing">shirt</name>
</item>
```

has two datatypes whose full names are "urn:uuid:C2F41010-65B3-11d1-A29F-00AA00C14882/integer" and

"http://zoosports.com/dt?clothing".

What a datatype's URI Means

Datatypes are identified by URIs. The URI as simply a reference to a section of a document that defines the appropriate parser and storage format of the element. To make this broadly useful, this document defines a set of common data types including all common forms of dates, plus all basic datatypes commonly used in SQL, C, C++, Java and COM (including strings).

The best form of such a document is that it should itself be an XML-Data schema where each datatype is an element declaration. For this purpose we define a <*Syntax*> subelement which can be used in lieu of a content model. We also define an <*objecttype*> subelement. Each has a URI as its value. This integrates data types with element types in general.

```
<schema:elementType id="int">
  <syntax  href="urn:uuid:C2F41010-65B3-11d1-A29F-00AA00C14882/num_to_int"
  <objectType href="urn:uuid:C2F41010-65B3-11d1-A29F-00AA00C14882/integer32
</schema:elementType>

<schema:elementType id="date.iso8601">
  <syntax  href="urn:uuid:C2F41010-65B3-11d1-A29F-00AA00C14882/date.iso8601
  <objecttype href="urn:uuid:C2F41010-65B3-11d1-A29F-00AA00C14882/integer32
</schema:elementType>
```

The objecttype sub-element can reference a structure, Java class, COM++ coClass, etc. The syntax subelement identifies a parser which can decode the element's content (and/or attributes) into the object type given the storage type URI. Input to the parser is the element object exposing all its attributes and content tree (that is, the subtree of the grove beginning with the element containing the dt attribute). The objectType attribute in particular is assumed available to the parser so that a single parser can support several objecttypes.

Having said this, all *basic* data types should be built into the parsers for efficiency and in order to ground the process. For these, the datatype elements serve only to formally document the storage types and parsers, and to give higher-level systems (such as RDF) a more formal basis for datatypes.

I do not currently propose that we attempt to write any universal notation for parsing rules. Certain popular kinds of formats, particularly dates, are not easily expressed in anything but natural language or code, and the parsers must be custom written code. In other words, the URIs for the basic syntax and objecttype elements probably resolve only to text descriptions.

Structured Data Type Attributes

Attributes in cannot XML have structure. I will separately propose some techniques to avoid this problem, specifically that the XML API should contain a method that treats attributes and subelements indistinguishably, and also that the content which is an element's value can be syntactically separated from content which is an element's properties.

Specific Datatypes

This includes all highly-popular types and all the built-in types of popular database and programming languages and systems such as SQL, Visual Basic, C, C++ and Java(tm).

Name	Parse type	Storage type	Examples

`string`	pcdata	string (Unicode)	Omwnuma legatai wn onoma monon koinon, o de kata tounoma logos thV ousiaV eteros, oion zuon o te anqropoV kai to gegrammenon.
`number`	A number, with no limit on digits, may potentially have a leading sign, fractional digits, and optionally an exponent. Punctuation as in US English.	string	15, 3.14, -123.456E+10
`int`	A number, with optional sign, no fractions, no exponent.	32-bit signed binary	1, 58502, -13
`float`	Same as for "number."	64-bit IEEE 488	.314159265358979E+1
`fixed.14.4`	Same as "number" but no more than 14 dights to the left of the decimal point, and no more than 4 to the right.	64-bit signed binary	12.0044
`boolean`	"1" or "0"	bit	0, 1 (1=="true")
`dateTime.iso8601`	A date in ISO 8601 format, with optional time and no optional zone. Fractional seconds may be as precise as nanoseconds.	Structure or object containing year, month, hour, minute, second, nanosecond.	19941105T08:15:00301
`dateTime.iso8601tz`	A date in ISO 8601 format, with optional	Structure or object containing	19941105T08:15:5+03

	time and optional zone. Fractional seconds may be as precise as nanoseconds.	year, month, hour, minute, second, nanosecond, zone.	
`date.iso8601`	A date in ISO 8601 format. (no time)	Structure or object containing year, month, day.	19541022
`time.iso8601`	A time in ISO 8601 format, with no date and no time zone.	Structure or object exposing day, hour, minute	
`time.iso8601.tz`	A time in ISO 8601 format, with no date but optional time zone.	Structure or object containing day, hour, minute, zonehours, zoneminutes.	08:15-05:00
`i1`	A number, with optional sign, no fractions, no exponent.	8-bit binary	1, 255
`i2`	"	16-bit binary	1, 703, -32768
`i4`	"	32-bit binary	
`i8`	"	64-bit binary	
`ui1`	A number, unsigned, no fractions, no exponent.	8-bit unsigned binary	1, 255
`ui2`	"	16-bit unsigned binary	1, 703, -32768
`ui4`	"	32-bit unsigned binary	

ui8	"	64-bit unsigned binary	
r4	Same as "number."	IEEE 488 4-byte float	
r8	"	IEEE 488 8-byte float	
float.IEEE.754.32	"	IEEE 754 4-byte float	
float.IEEE.754.64	"	IEEE 754 8-byte float	
uuid	Hexidecimal digits representing octets, optional embedded hyphens which should be ignored.	128-bytes Unix UUID structure	F04DA480-65B9-11d1-A29F-00AA00C14882
uri	Universal Resource Identifier	Per W3C spec	http://www.ics.uci.edu/pub/ietf/uri/draft-fielding-uri-syntax-00.txt http://www.ics.uci.edu/pub/ietf/uri/ http://www.ietf.org/html.charters/urn-charter.html
bin.hex	Hexidecimal digits representing octets	no specified size	
char	string	1 Unicode character (16 bits)	
string.ansi	string containing only ascii characters <= 0xFF.	Unicode or single-byte string.	This does not look Greek to me.

All of the dates and times above reading "iso8601.." actually use a restricted subset of the formats defined by ISO 8601. Years, if specified, must have four digits. Ordinal dates are not used. Of formats employing week numbers, only those that truncate year and month are allowed (5.2.3.3 d, e and f).

Mapping between Schemas

Certain uses of data emphasize syntax, others "conceptual" relations. Syntactic schemas often have fewer elements

compared to explicitly conceptual ones. Further, it is usually easier to design a schema that merely covers syntax rather than designing a well-thought-out conceptual data model. An effect of this is that many practical schemas will not contain all the elements that a conceptual schema would, either for reasons of economy or because the initial schema was simply syntactic. But is it useful to make the implicit explicit over time so that more generic processors can make use of data.

For example, the following schema is essentially syntax:

```
<elementType id="author">
    <string/>
</elementType>

<elementType id="title">
    <string/>
</elementType>

<elementType id="Book">
    <element type="#title"/>
    <element type="#author" occurs="ONEORMORE"/>
</elementType>
```

with instances looking like this

```
<Book>
    <title>Paradise Lost</title>
    <author>Milton</author>
</Book>
```

On the other hand, a conceptual schema could look like this:

```
<elementType id="name">
    <string/>
</elementType>

<elementType id="Person">
    <element type="#name/>
</elementType>

<elementType id="creator">
    <range type="#Person/>
</elementType>

<elementType id="title">
    <string/>
</elementType>

<elementType id="Book">
    <element type="#title"/>
    <element type="#creator" occurs="ONEORMORE"/>
</elementType>
```

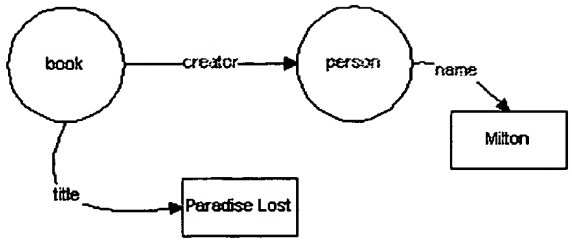
If fully explicit, its instances would look something like this:

```
<Person id="thing1">
     <name>Milton</Person>
</Person>

<Book>
```

```
    <title>Paradise Lost</title>
    <creator>
        <Person>
            <name>Milton</name>
        </Person>
    </creator>
</Book>
```

In any case, what we want to express is a diagram such as this:



To do this, we will add mapping information into the syntactic schema which tells us how to interpolate the implied elements (and also to map *author* to *creator*) thereby creating a conceptual data model.

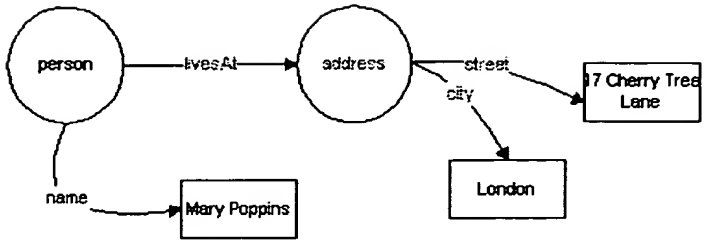
```
<?xml:namespace href="uri-to-the-conceptual-schema" as="c" ?>
<elementType id="author">
    <string/>
</elementType>

<elementType id="title">
    <string/>
</elementType>

<elementType id="Book">
    <mapsTo type="c:book"/>
    <element type="#title"> <mapsTo type="c:title"/> </element>
    <element type="#author" occurs="ONEORMORE">
        <mapsTo type="string">
            <implies type="c:name">
                <implies type="c:person">
                    <implies type="c:creator"/>
                </implies>
            </implies>
        </mapsTo>
    </element>
</elementType>
```

A more complex case could involve needing to map several properties to have a common implied node. For example, suppose we wanted that a *street* element and *city* element should both imply the same *address* node.

```
<Person>
    <name>Mary Poppins</name>
    <street>17 Cherry Tree Lane</street>
    <city>London</city>
</Person>
```



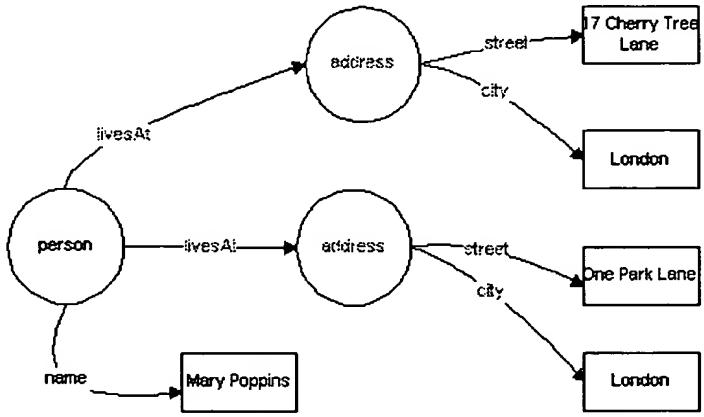
That is, rather than creating two *address* nodes, we want to create only a single one, and subordinate both the *street* and *city* to it. If the conceptual schema has elements *livesAt*, *address*, *street* and *city*, we could write a mapping thus:

```
...definitions of name, street and city...

<elementType id="Person">
    <mapsTo type="c:person"/>
    <element type="#name">    <string/>   <mapsTo type="c:name"/>   </eleme
    <element type="#street">
        <string/>
        <mapsTo type="c:street">
              <implies type="c::address" id="livesAtAddress">
                  <implies type="c:livesAt"/>
              </implies>
        </mapsTo>
    </element>
    <element type="#city">
        <string/>
        <mapsTo type="c:city">
            <implies type="#livesAtAddress"/>
        </mapsTo>
    </element>
</elementType>
```

Elements may be repeated, so mapping rules need to accommodate repetitions. Suppose that someone has two addresses in the grammatical syntax, this needs to map to two addresses in the graph while still keeping the structure correct.

```
<Person>
    <name>Mary Poppins</name>
    <street>17 Cherry Tree Lane</street>
    <city>London</city>
    <street>One Park Lane</street>
    <city>London</city>
</Person>
```



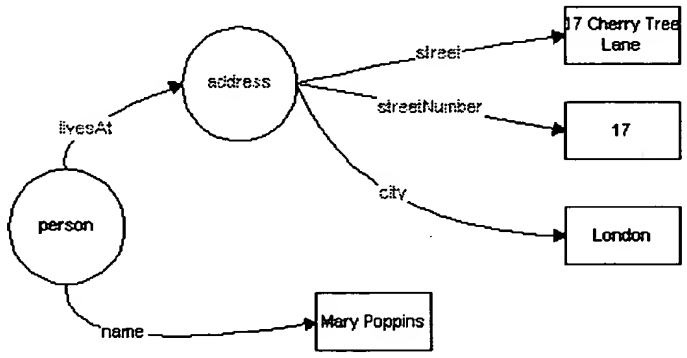
```
<elementType id="Person">
    <mapsTo type="c:person"/>
    <element type="#name">    <string/>    <mapsTo type="c:name"/>    </eleme
    <group occurs="ZEROORMORE"/>
        <element type="#street">
            <string/>
            <mapsTo type="c:street">
                <implies type="c::address" id="livesAtAddress">
                    <implies type="c:livesAt"/>
                </implies>
            </mapsTo>
        </element>
        <element type="#city">
            <string/>
            <mapsTo type="c:city">
                <implies type="#livesAtAddress"/>
            </mapsTo>
        </element>
    </group>
</elementType>
```

Mappings within groups are handled together. Since *street* and *city* are in a single group, each occurrence of such a group results in one *address*.

Text markup can also be handled by mapping. Suppose that for some reason we choose to markup the number portion of a street address:

```
<Person>
    <name>Mary Poppins</name>
    <street>< streetNumber>17</ streetNumber > Cherry Tree Lane</street>
    <city>London</city>
</Person>
```

If this should be reflected in the graph,



We can do that with mapping such as:

```
<elementType id="streetNumber">
    <string/>
</elementType>

<elementType id="street>
    <mixed>
        <element type="# streetNumber">
            <mapsTo type="c: streetNumber">
```

```
                    <implies type="#livesAtAddress"/>
                </mapsTo>
            </element>
        </mixed>
    </elementType>

    ...Person defined as before...
```

Appendix A: Examples

Some data:

```
<?xml:namespace name="http://company.com/schemas/books/" as="bk"/>
<?xml:namespace name="http://www.ecom.org/schemas/dc/" as="ecom" ?>

<bk:booksAndAuthors>
    <Person>
        <name>Henry Ford</name>
        <birthday>1863</birthday>
    </Person>

    <Person>
        <name>Harvey S. Firestone</name>
    </Person>

    <Person>
        <name>Samuel Crowther</name>
    </Person>

    <Book>
        <author>Henry Ford</author>
        <author>Samuel Crowther</author>
        <title>My Life and Work</title>
    </Book>

    <Book>
        <author>Harvey S. Firestone</author>
        <author>Samuel Crowther</author>
        <title>Men and Rubber</title>
        <ecom:price>23.95</ecom:price>
    </Book>
</bk:booksAndAuthors>
```

The schema for http://company.com/schemas/books:

```
<?xml:namespace name="urn:uuid:BDC6E3F0-6DA3-11d1-A2A3-00AA00C14882/" as="s"/?
<?xml:namespace href="http://www.ecom.org/schemas/ecom/" as="ecom" ?>

<s:schema>

    <elementType id="name">
        <string/>
    </elementType>

    <elementType id="birthday">
        <string/>
        <dataType dt="date.ISO8601"/>
```

```
    </elementType>

    <elementType id="Person">
        <element type="#name" id="p1"/>
        <element type="#birthday" occurs="OPTIONAL">
            <min>1700-01-01</min><max>2100-01-01</max>
        </element>
        <key id="k1"><keyPart href="#p1" /></key>
    </elementType>

    <elementType id="author">
        <string/>
        <domain type="#Book"/>
        <foreignKey range="#Person" key="#k1"/>
    </elementType>

    <elementType id="writtenWork">
        <element type="#author" occurs="ONEORMORE"/>
    </elementType>

    <elementType id="Book" >
        <genus type="#writtenWork"/>
        <superType href=" http://www.ecom.org/schemas/ecom/commercialItem"/>
        <superType href=" http://www.ecom.org/schemas/ecom/inventoryItem"/>
        <group groupOrder="SEQ" occurs="OPTIONAL">
            <element type="#preface"/>
            <element type="#introduction"/>
        </group>
        <element href="http://www.ecom.org/schemas/ecom/price"/>
        <element href="ecom:quantityOnHand"/>
    </elementType>

    <elementTypeEquivalent id="livre" type="#Book"/>
    <elementTypeEquivalent id="auteur" type="#author"/>

</s:schema>
```

Appendix B : An XML DTD for XML-Data schemas

```
<!ENTITY % nodeattrs 'id ID #IMPLIED'>

<!-- href is as per XML-LINK, but is not required unless there is
      no content -->

<!ENTITY % linkattrs
                    'id ID #IMPLIED
                     href CDATA #IMPLIED'>

<!ENTITY % typelinkattrs
                    'id ID #IMPLIED
                     type CDATA #IMPLIED'>

<!ENTITY % exattrs
                    'name CDATA #IMPLIED
                     content (OPEN|CLOSED) "OPEN" >

<!ENTITY % elementTypeElements1
                    'genus? correlative? superType*'>
```

```
<!ENTITY % elementTypeElements2
                     'description,
                      (min|minExclusive)?,
                      (max | maxInclusive)?,
                      domain*,
                      key*,
                      foreignKey*,
                      (datatype | ( syntax?, objecttype+ ) )?
                      mapsTo?'>

<!ENTITY % elementConstraints
                     'min? max? default?'>

<!ENTITY % elementAttrs
                     'occurs (REQUIRED|OPTIONAL|ONEORMORE|ZEROORMORE) "REQUIRE

<!ENTITY % rangeAttribute
                     'range CDATA #IMPLIED'  >

<!-- The top-level container -->
<!element schema     ((elementType|linkType|
                       extendType|
                       intEntityDcl|extEntityDcl|
                       notationDcl|extDcls)*)>
<!attlist schema %nodeattrs;>

<!-- Element Type Declarations -->

<!element elementType (%elementTypeElements1;,
                       ((element|group)*|empty|any|string|mixed)?,
                        attribute*
                        %elementTypeElements2 )>

<!attlist elementType %nodeattrs;
                      %exattrs >

<!-- Element types allowed in content model -->

<!-- Note this is just short for a model group with only one element in it -->
<!element element  (%elementConstraints;) >

<!-- The type is required -->
<!attlist element   %typelinkattrs;
                    %elementAttrs;
                    presence (FIXED) #IMPLIED >

<!-- A group in a content model: and, sequential or disjunctive -->
<!element group     ((group|element)+)>
<!attlist group     %nodeattrs;
                    %elementattrs;
                    presence (FIXED) #IMPLIED
                    groupOrder (AND|SEQ|OR) 'SEQ'>

<!element any       EMPTY>
<!element empty     EMPTY>
<!element string    EMPTY>

<!-- mixed content is just a flat, non-empty list of elements -->
<!-- We don't need to say anything about <string/> (CDATA), it's implied -->
```

```
<!element mixed          (element+)>
<!attlist mixed          %nodeattrs;>

<!element superType  EMPTY>
<!attlist superType %linkattrs;>

<!element genus  EMPTY>
<!attlist genus %typelinkattrs;>

<!element description  MIXED>
<!attlist description %nodeattrs;>

<!element domain  EMPTY>
<!attlist domain %typelinkattrs;>

<!element default MIXED>
<!attlist default %nodeattrs;>

<!element min MIXED>
<!attlist min %nodeattrs; >

<!element max MIXED>
<!attlist max %nodeattrs; >

<!element maxInclusive MIXED>
<!attlist maxInclusive %nodeattrs; >

<!element minExclusive MIXED>
<!attlist minExclusive %nodeattrs; >

<!element key (keyPart+)>
<!attlist key %nodeattrs;>

<!element keyPart EMPTY>
<!attlist keyPart %linkattrs;>

<!element foreignKey foreignKeyPart* >
<!attlist foreignKey %nodeattrs;
                     %rangeAttribute;
                     key CDATA #IMPLIED >

<!element foreignKeyPart EMPTY>
<!attlist foreignKeyPart %linkattrs;>

<!-- Datatype support -->

<!element datatype (elementType?) >
<!attlist datatype %typelinkattrs;
       presence (IMPLIED|SPECIFIED|REQUIRED|FIXED) #IMPLIED >

<!element syntax >
<!attlist syntax %linkattrs; >

<!element objecttype >
<!attlist objecttype %linkattrs; >


<!-- Mapping support -->
```

```
<!element mapsTo (implies?)>
<!attlist mapsTo %typelinkattrs;>

<!element implies (implies?)>
<!attlist implies %typelinkattrs;>

<!-- Alias support -->


<!element elementTypeEquivalent EMPTY>
<!attlist elementTypeEquivalent %typelinkattrs; >

<!element correlative EMPTY>
<!attlist correlative %linkattrs;>


<! Subtype of ElementType that is explicitly a relation. -->

<!element relationType (%elementTypeElements1;,
                        ((element|group)*|empty|any|string|mixed)?,
                        attribute*
                        %elementTypeElements2)>
<!attlist relationType %nodeattrs;
                   %exattrs; >



<!-- Attributes -->
<!-- default value must be present if presence is specified or fixed -->
<!-- presence defaults to specified if default is present, else implied -->

<!element attribute  (%PropertyElements1,
                      %PropertyElements2,
                      %elementConstraints)>
<!attlist attribute  %typelinkattrs;
                     name CDATA #IMPLIED
                     %elementAttrs
                     dt          CDATA #IMPLIED
                     atttype    (URIREF|
                                 ID|IDREF|IDREFS|ENTITY|ENTITIES|
                                 NMTOKEN|NMTOKENS|
                                 ENUMERATION|NOTATION|CDATA) CDATA
                     %rangeAttribute;
                     default CDATA #IMPLIED
                     values NMTOKENS #IMPLIED
                     presence (IMPLIED|SPECIFIED|REQUIRED|FIXED) #IMPLIED >

<!-- Notation and Entity Declarations -->
<!-- Note: as this is written, only external entities
      can have structure without escaping it -->
<!-- 'par' is TRUE iff parameter entity. -->
<!-- systemID and publicId (if present) must have the required syntax -->

<!ENTITY % notationattrs '%nodeattrs
                          systemID CDATA #IMPLIED
                          publicID CDATA #IMPLIED'>

<!ENTITY % entityattrs   '%notationattrs
                          name CDATA #IMPLIED
                          par     (TRUE | FALSE) "FALSE">
```

```
<!-- Notation Declarations -->

<!element notationDcl        EMPTY>
<!attlist notationDcl        %notationattrs>

<!element intEntityDcl       PCDATA>
<!attlist intEntityDcl       %entityattrs; >

<!-- The entity will be treated as binary if a notation is present -->
<!element extEntityDcl       EMPTY>
<!attlist extEntityDcl       %entityattrs;
                             notation CDATA #IMPLIED>

<!-- External entity with declarations to be included -->
<!element extDcls            EMPTY>
<!attlist extDcls            %entityattrs;>
```